UNIVERSITY OF OSLO
Department of Informatics

Dialog Act Recognition using Dependency Features

Master's thesis

Sindre Wetjen

November 15, 2013

Acknowledgments

First I want to thank my supervisors **Lilja Øvrelid** and **Pierre Lison** for their time, effort and guidance. It has been a real privilege to work with such talented, knowledgeable and friendly people.

I also want to tank my fellow students **Trond Thorbjørnsen**, **Emanuele Lapponi**, **Arne Skjærholt** and the rest of the students on the 7th floor for coffee breaks, discussions and encouragement when the road ahead seemed long.

I am grateful to the UiO Language Technology Group for creating an including learning environment.

Last, but not least, I want to thank **Line Moseng** for helping me keep track of night and day.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> "The literature of the fantastic abounds in inanimate objects magically endowed with sentience and the gift of speech. From Ovid's statue to Mary Shelley's Frankenstein, there is something deeply touching about creating something and then having a chat with it." – Jurafsky and Martin (2009, p. 847)

The dream about talking to inanimate objects is not something that computational linguists have discarded. Being able to talk and interact with our computers is the main goal of a dialog system. For it to happen the computer has to not only be able to produce words out of the sound coming through the air, it also has to understand, create a response and reply.

Many speech-based interfaces have, instead of being geared towards understanding our every day spoken language, focused on delivering a command-like language that you can use to query devices via voice. But to be able to make a conversation we need to enable our computer to understand us using the same spoken language that we use between humans. Consider the following conversation:

1) Mario: Hi!

2) Luigi: Hi, how are you?

3) Mario: I'm fine, how about you?

4) Luigi: Oh, you know, working hard.

5) Mario: Yeah, such a shame to have to work so hard when the weather is so nice.

In a natural language understanding system, this is most commonly done by reducing our utterances into more abstract concepts about what information these utterances express. One such abstraction, in this case a high-level

view of the conversation structure, can be applied to the small section from the beginning of a conversation between Mario and Luigi shown above. Transcribing the conversation using dialog acts we can see how this conversation can be viewed.

1) Mario: Hi! [open]

2) Luigi: Hi [open], how are you [open_question]?

3) Mario: I'm fine [answer], How about you [open_question]?

4) Luigi: Oh, you know, working hard [answer].

5) Mario: Yeah [affirm], such a shame to have to work so hard when the weather is so nice [opinion].

The conversation starts with an opening from Mario, it then continues with an opening from Luigi which in turn asks a question. The question is answered by Mario and a new question is asked. That question continues the conversation by requiring Luigi to answer. Luigi then does not ask a new question, but what Mario does is affirm that the answer is received. Mario does this simply by saying "yeah". The last line in our conversation also contains an opinion, namely that Mario thinks working hard while the weather is nice is a shame.

The conversation between Mario and Luigi would probably have continued beyond the small section until one of them said something that is considered a closing of the conversation. In the meanwhile the information that is shared between them grows with each uttering. With the necessity to handle that information in a good way inside the machine, the abstraction can be very useful.

This thesis focuses on the problem of automatically extracting such pragmatic abstractions from the raw utterances. The task is often referred to as dialog act classification.

1.1 Thesis

This thesis aims to contribute in the on-going task of dialog act recognition for use in general purpose Dialog Systems. We propose a dialog act classification system using machine learning and features extracted from syntactic representations, more specific dependency representations.

The main purpose of the thesis is to investigate whether dependency features can improve the accuracy of a dialog act recognizer. More specifically, we compare a dialog act recognizer using no syntactically informed features against three classifiers integrating features derived from the dependency tree of the utterance to classify.

To extract these features we train a parser on spoken language data. We furthermore investigate whether a parser trained on spoken language differs from one trained on written language and if incorporating some of the spoken language phenomena improves the classification task. We do this by converting a phrase-structure treebank to a dependency treebank using an "off-the-shelf" converter. We then propose an algorithm for post-processing the dependency treebank to include some spoken language phenomena.

In order to investigate whether the syntactic trees improve the classification task we develop a Dialog Act Recognition system. We then compare an instance of this system using no syntactically informed features against three other versions of our dialog act recognition system with syntactic features: one with a parser trained on the Wallstreet Journal Treebank of written text, one trained on a spoken language treebank with no extra annotation and one which was produced using the algorithm.

1.1.1 Thesis Structure

The thesis is structured as follows:

Chapter 2

This chapter provides an introduction to the theories and background that the reader would need to understand the work described in the thesis. The chapter will touch on topics like Dialog Systems, Phrase-Structure and Dependency Grammars, Dialog Acts, Machine Learning and more.

Chapter 3

This chapter describes how we created our Treebank for Spoken Language. It will describe the procedure that we propose to create a dependency treebank for spoken language and explain in detail how it works. We go on to train a dependency parser on this data and compare this parser against three other parsers with two more Switchboard treebanks and one trained on the Wallstreet Journal treebank.

Chapter 4

Chapter 4 is about the dialog act classification task mentioned above. The chapter proposes a set of baseline features and syntactic features using trees from the parsers described in Chapter 3. The end of the chapter consists of a detailed comparison of the baseline system and the system extracting features from the parser trained on the treebank created in Chapter 3.

Chapter 5

In chapter 5 we discuss some of the results and conclusions that can be made on the basis of the results found in chapter 3 and 4. We will also peek at some of the future work that might help improve the combination of dependency parsers and dialog act classification.

Chapter 2

Background

This chapter will focus on giving the reader an overview of the topics used as a basis for Dialog Act Recognition and Syntactic Parsing of Spoken Language. We will introduce Dialog Systems, which dialog act classification is most commonly used in and where our classifier fits in the broader picture.

The first section, the section on Dialog Systems, will give an overview of what a dialog system contains. The section Dialog Acts will give an overview what the content of dialog acts usually is. The last two sections on Syntactic Parsing and Spoken Language are brief introductions into the field of Syntactic parsing of natural language using computers. A lot of the inspiration and references for this work is taken out from the works of Jurafsky and Martin's book on Speech and Language Processing (Jurafsky & Martin, 2009, p. 847 – 894).

2.1 Dialog Systems

Dialog systems are systems designed to keep a conversation with a user. This includes the entire process from taking speech input in the form of sound waves, make a decision and respond appropriately. The task is big and like most big tasks the divide and conquer strategy is applied to solve the task.

Figure 2.1 shows a common way of dividing dialog systems into different modules (Jurafsky & Martin, 2009; Young, 2002). The arrows shows the way the information is flowing and the order the components work in, from user input to user feedback. Each of these components has been the subject of considerable research and depend on each other to produce as good and accurate result as possible in order to achieve the end goal, talking to you.

What this thesis will focus on is the Natural Language Understanding part of the system (the green box in 2.1), so we will in this section present an overview of all the components shown in figure 2.1 with a special emphasis on how they interact with the natural language understanding component and the component itself.

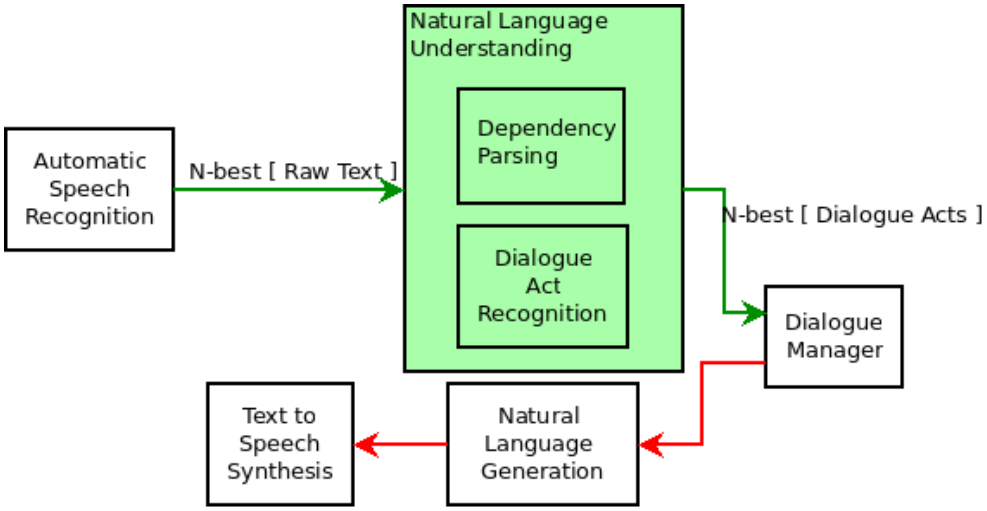

Figure 2.1: An overview of how a dialog system is commonly designed.

2.1.1 Automatic Speech Recognition

The Automatic Speech Recognition (ASR) component marks the beginning of the system's processing pipeline; it receives sound input from a user via recording equipment. The ASR is the component responsible for taking the speech signal and producing the text that corresponds to that sound. Figure 2.2 gives an overview of this process, showing the microphone giving a speech signal to the ASR and the ASR producing a list of possible utterances that the pattern in the speech signal matches. There are many problems that are related to this process, and ASRs are known to be particularly error-prone, which means words can be dropped or misheard. This is even a problem for humans, so it would be unreasonable to expect an ASR to be 100% correct all the time.

Another problem pertaining to speech and ASR is to determine where an utterance starts or ends. This is not trivial as speakers can take turns in a very tight sequence, with typically very small gaps between turns.

Being the component in front of the Natural Language Understanding

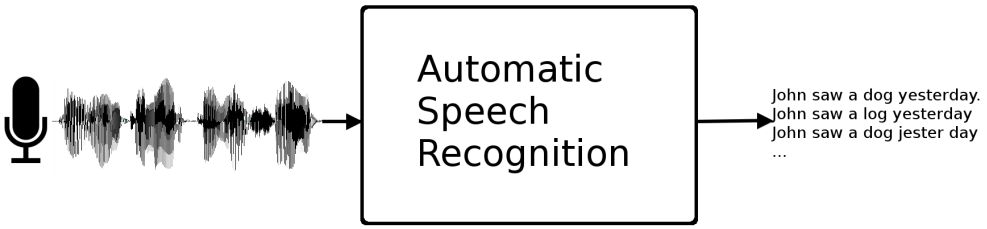

Figure 2.2: Details of the input output to/from a Automatic Speech Recognition unit.

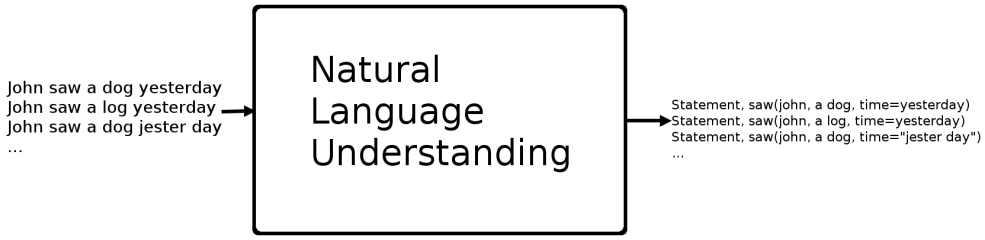

Figure 2.3: Details of the input output to/from a Natural Language Understanding unit.

(NLU) component means that the output of the ASR component is the input of the NLU component. All the problems the ASR does not cope with, the NLU has to handle in some way or another. This close bond is reflected in that many tasks are defined to be in one or the other component depending on the system. And indeed in some systems, like in Young (2002), there is only one component called Language Understanding incorporating all the problems for the NLU and the ASR.

2.1.2 Natural Language Understanding

The Natural Language Understanding component comes in many shapes and forms depending on the domain and the level of understanding that is required for fulfilling the systems purpose and reacts the way a user expects. Figure 2.3 shows what the NLU component should ideally do, map the input from the Automatic Speech Recognition to a semantic and pragmatic interpretation of the utterance. In this figure, the input is a list of hypotheses from the ASR mapped to a dialog act, as described in Section 2.4, Dialog Acts, and a representation of the meaning found in the sentence.

Dialog systems were (and still are) used in very domain specific ways. By domain specific we mean that a system has to cover a limited subset of all possible utterances in a language that are relevant to solving one specific task. E.g. ordering flight tickets or virtual switchboards with interactive voice response functions.

These kinds of systems often use a Natural Language Understanding component that is very simple and does not parse more utterances than the domain it handles. The representation of the utterances are also very shallow, and does only what is absolutely necessary to fill in the obligatory slots for the dialog manager to make a decision for its limited domain. Examples include the frame-and-slot based GUS system from as far back as 1977 (Bobrow et al., 1977) and the semantic HMM models of Pieraccini and Levin (1992).

Another approach to deal with the amount of utterances the system has to handle is by instead of requiring the system to understand any utterance that is normal for human-to-human interaction, the system requires the users

to speak in certain ways so that the system has an easier time of understanding what the user wants. This approach is shown in systems like CommandoTalk (Stent, Dowding, Gawron, Bratt, & Moore, 1999) or voice interfaces to search engines (Schalkwyk et al., 2010). The problem with this approach is that while the natural language understanding might be easier if you give the users predefined frames to work into it is not going to feel like a fluent two-way conversation for the user. In short, the user has to learn or understand the system in order to use it instead of the other way around.

Constraining the speaker to predefined templates does not make for a natural conversation between the human and the machine, and filling in frames defined by the domain of the system does not make for a general purpose query to a system. To enable a system to take queries in a natural form from a user and scale it automatically beyond one task or language constraints, the NLU component has to do a lot more.

This thesis looks at how we might be able to give the user an interface with natural language and how syntactic parsing may help. More specifically we will investigate how dialog act classification can be improved with the help of syntactic features extracted via a data-driven dependency parser in the Natural Language Understanding component. These concepts will be introduced in Section 2.2.

2.1.3 Dialog Manager

The job of the Dialog Manager is to make decisions based on the information given by the NLU component and decide what to do on the background of this information. If the system is more than a simple question-answering system, the dialog manager has to keep track of where the conversation has been and which information has been ascertained from the dialog acts coming in, what is uncertain and needs verification and what the system still needs to know.

The main interaction method between the user and the system is through talking to it, and the dialog manager must therefore select the systems action to perform on the basis of the interpreted user inputs. Parsing the speech signal correctly is for this reason very important for the dialog manager to make the correct decision. It is also important that the input is as feature rich as possible so that the dialog manager can make informed decisions about the state and intentions of the user (Young, 2002; Jurafsky & Martin, 2009).

2.1.4 Natural Language Generation & Text-to-Speech Synthesis

The speech understanding part of the system is not directly affected or influenced by the Natural Language Generation or Text-to-Speech Synthesis components, since their purpose is producing the response that the Dialog

Manager decides is appropriate. They are nonetheless an important part of a Dialog System.

The task is to receive dialog acts from the Dialog Manager which has made a decision and wants the user to receive a response from the system. The natural language generation component takes this act, produces a sentence in a natural language that reflects the dialog acts intent; the sentence is then handed over to the Text-to-Speech Synthesis. The Text To Speech Synthesis component then converts the utterance to a speech signal so the user can hear the response.

2.2 Syntactic Parsing

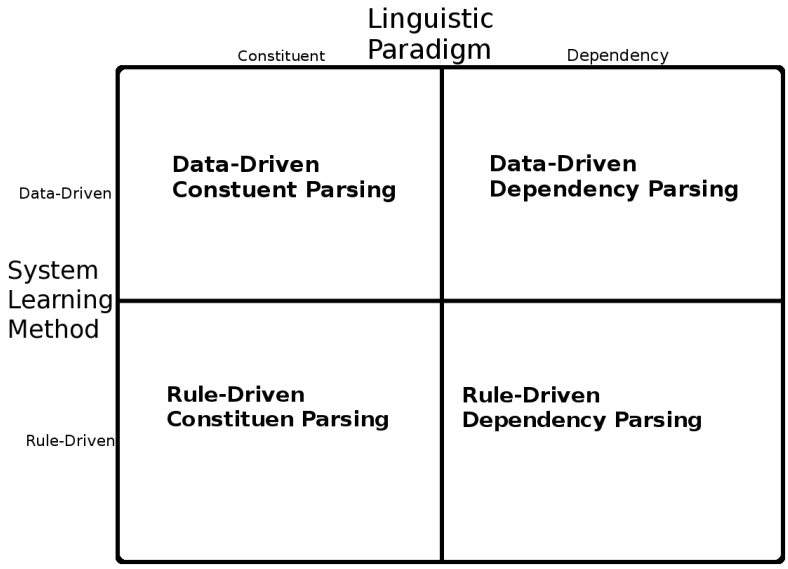

Figure 2.4: An overview of the different paradigms used in Syntactic Parsing systems.

Syntactic Parsing is a field of informatics that has a long history, with papers on machine translation going as far back as the mid 1930s and ranging in complexity from pattern matching to multi-layered rule-based systems. We will not go in depth on the whole history and usage of Syntactic Parsing, but touch on the different concepts and describe some of the relevant parts of Syntactic Parsing and its underpinning linguistic theories in this section.

Figure 2.4 is an overview of the major categories of current approaches for syntactic and semantic parsing. The horizontal axis shows the syntactic framework that are most commonly used in computer representations of syntax today and the vertical axis shows the method used to make the representation. We will briefly describe both of the syntactic frameworks and theX learning

methods. Then we describe some pros and cons. Then we take a more in-depth look at the paradigm that the system in this thesis uses, namely Data-Driven Dependency Parsing.

2.2.1 Phrase Structure Grammar

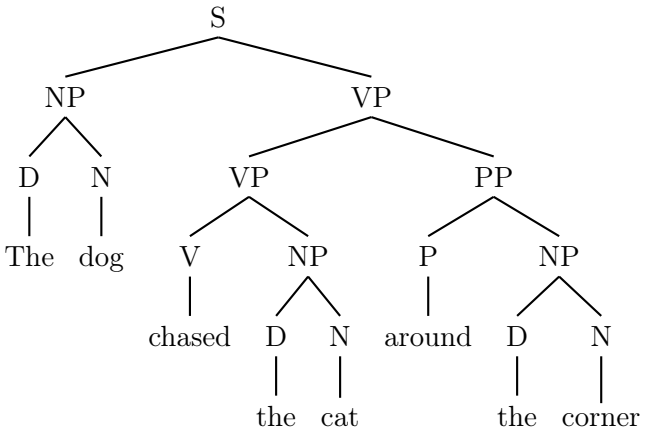

Figure 2.5: A Phrase Structure Tree of our example sentence "The dog chased the cat around the corner"

$$
\begin{aligned}
&S \rightarrow NP\ VP \\
&NP \rightarrow D\ N \\
&VP \rightarrow V\ NP\ /\ VP\ PP \\
&PP \rightarrow P\ NP \\
\\
&D \rightarrow the\ /\ a \\
&N \rightarrow dog\ /\ cat\ /\ corner \\
&V \rightarrow chased \\
&P \rightarrow around
\end{aligned}
$$

Figure 2.6: Context-Free Grammar that builds the tree in figure 2.5.

The first syntactic framework we are going to take a look at is Phrase Structure Grammar. Phrase Structure Grammar was conceived as an idea by Wilhelm Wundt (1900), but was first formalized by the linguist Noam Chomsky in 1956. A phrase structure grammar builds on the notion of a hierarchical structure based on the phrase structures found in a snentence. Figure 2.5 shows a phrase structure tree, which illustrates this hierarchy with nouns and determiners combining into a noun phrase etc.

We look at the Syntactic Framework of Phrase Structure Grammars because the data that we will work with in this thesis are based on Phrase Structure Grammar trees like the one shown in 2.5. Also, Context-Free Grammar (CFG) serves as a good steppingstone to explain the difference between rule and data driven systems and how the syntactic frameworks differ.

Context-Free Grammar

Context-Free Grammar (CFG) is a formalized phrase structure grammar that is the basis for some of the theories used in modern parsers. A short example grammar is displayed in Figure 2.6 for the reader.

The grammar in Figure 2.6 shows that phrase structures in a CFG are named on the left side of the arrow. These are called Non-Terminals. Terminals are the surface level tokens, written in lowercase. The right side consists of a mix of Terminals and Non-Terminals that makes up the named phrase structure. There exists a lot of variants of context-free grammars with different restrictions, but for the remainder of this thesis we will stick to the general notion of a CFG as described by Jurafsky and Martin(Jurafsky & Martin, 2009). A grammar G is defined by four parameters N, E, R S.

- N a set of *non-terminal* symbols.
- E a set of *terminal* symbols disjoint from N.
- R a set of *rules* or productions in the form A →b where b is a string from the infinite set of string (EuN)*
- S a designated *start symbol*

from the grammar in Figure 2.6, the different categories take the following values:

- N: {S, NP, VP, PP, D, N, V, P}
- E: {the, a, dog, cat, corner, chased, around}
- R: {S → NP VP, ..., V → chased, ...}
- S: {S}

Part-of-Speech Our Context-Free Grammar in Figure 2.6 is a lexicalised Context-Free Grammar, meaning that the words are a part of the grammar. This may not be the case in all Syntactic Parsing systems. The words are instead labeled by their word category, and this is what we call a Part-of-Speech tag.

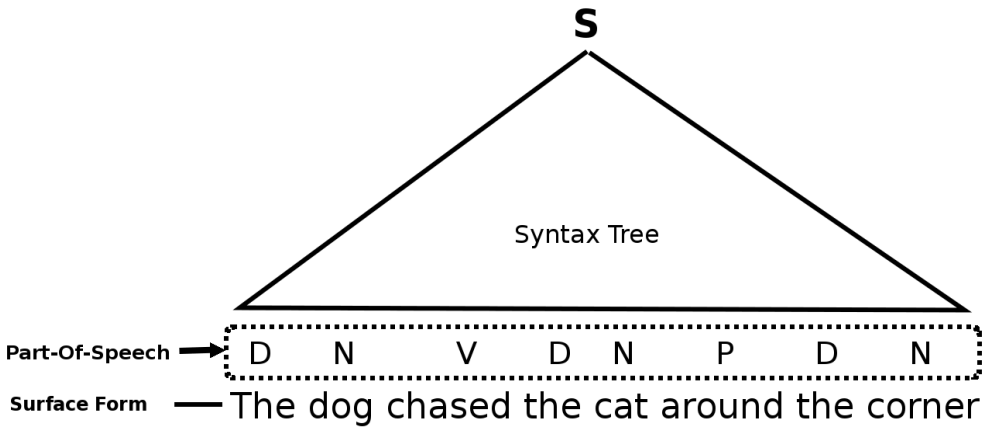

Figure 2.7: An illustration of what part of the system is called the Part-of-Speech (POS).

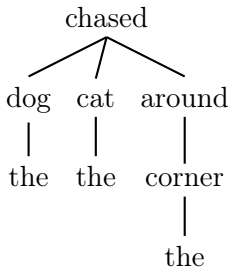

Figure 2.8: A Dependency Grammar tree made from the same sentence as Figure 2.5.

This task is often assigned to a Part-of-Speech tagger, and the Grammar then use the tags to make their parse trees. Figure 2.7 shows how this separation works from the surface form to the tree.

Modern parsers often use a combination, using both Part-Of-Speech tags and surface form values for the tree generation. The parser we will use and present is one such parser.

2.2.2 Dependency Grammar

Dependency Grammar is another syntactic framework that defines a sentence structure in a different way than the Phrase Structure Grammar framework introduced in the previous section. Instead of building up a hierarchical structure of phrase structures, it has a structure of word-to-word relations. This is shown in the example tree in Figure 2.8. It shows a dependency tree for the same saentence as in Figure 2.5. Both trees shows the same sentence "The dog chased the cat around the corner.".

The word-to-word relations are commonly described as *head* and *dependent*. The relation is said to be going from head to dependent. E.g In our example tree in Figure 2.8, the word "dog" is the *head* of "the" and a dependent of "chased".

Most modern notions of dependency grammar derive from the work done by Tesniere (1959), but the notion of word-to-word relations has its root as far back as the antiquity. Dependency Grammar failed to receive so much attention in the beginning of modern linguistics because it was considered by many to be inferior to its phrase structure counterpart. This was because of the mathematical analysis that Hays and Gaifman delivered on the properties of Dependency Grammar (Debusmann, 2000; Nivre, 2005). It has in later year received more attention because of its benefits when describing languages with a freer word order like Japanese, Latin, Russian and German where the projectivity requirements found in Hays and Gaifman Dependency Grammar (HGDG) are lifted.

Definitions of Dependency Grammar

There exist many formal definitions of Dependency Grammar that differ in some key aspects. This section will not go into all the details about the differences in the existing formalisms of dependency grammar. This should serve as an overview and be a good platform to understand how it differs from the phrase structure formalism described in the previous section.

Most of the formalisms in Dependency Grammar agree on three rules regarding the well-formedness of dependency trees. These are the rules of single-headedness, single root and acyclicity.

More formally, these three rules are described in Hays (1964); Gaifman (1965) Dependency Grammars interpreted by Nivre (2005) as the following set of rules:

1. For every w_i, there is at most one w_j such that $d(w_i, w_j)$.

2. For no w_i, $d^*(w_i, w_i)$

3. The whole set of word occurrences is connected by d.

The "*" in rule 2 denotes that it is a transitive relation. Rule 1 is the rule that defines single-headedness meaning that each word can have at most one head. Rules 1, 2 and 3 collectively ensures that the sentence is a well formed tree with a single root. Rule 2 is the acyclic rule.

The tree in Figure 2.8 is a good example of this. The tree is rooted in the word "chased". All the other words in the sentence is connected to the root by some path. Lastly, the tree contains no cycles and all paths goes directly to the root.

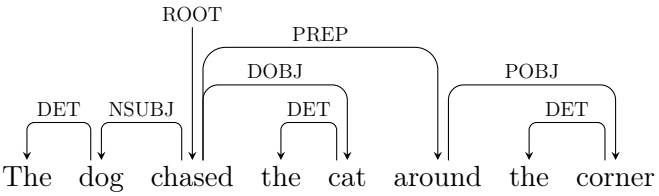

Figure 2.9: A different representation of a dependency tree for the sentence "The dog chased the cat around a corner." with arc labels.

Arc Labels

The idea of labels on the relations between the words is to describe the function that binds two words. This feature of Dependency Grammar is broadly adopted. Labels (or in some paradigms, functions) are the names placed above the arcs in Figure 2.9 which are not present in Figure 2.8.

The arc labels in the sample Figure 2.9 show a dependency graph where the arcs are labeled with its syntactic functions. The label "nsubj" shows that the relation between "dog" and "chased" is that the dog is the nominal subject of the sentence. The root, "chased", has a dependent which is the direct object ("dobj") and a prepositional modifier ("prep"). This is an example of a syntactic tree.

The labels does not have to be syntactic. Often it is more interesting to use semantic labels that will tell what is the action in the sentence, who is the agent and who is the patient rather than the syntactic relation between them.

A lot of the linguistic theories for dependency grammar has more than one set of label or arcs, arranged in a multi-stratal way, with different types of information, e.g syntactic and semantic. The frameworks and the parsing algorithms on the other hand are often mono-stratal (Nivre, 2005).

Projectivity

Projectivity is another imporant concept in Dependency Grammar. The projectivity rule is defined in the grammar proposed by Hays and Gaifman. It is defined as:

- If d*(w_i, w_j) and w_k is between w_i and w_j, then d*(w_k, w_j).

Roughly described as, if there is a transposed dependency relation between w_i and w_j and w_k is between w_i and w_j, that means there is a transposed dependency relation between w_k and w_j. In terms of graphs this means that at no point can there be crossing arcs inside the graph.

This feature restricts some dependency relations which are natural in some languages, and restricts the ways in which dependency grammars can elegantly account for relations like "John saw a dog yesterday which was a Yorkshire

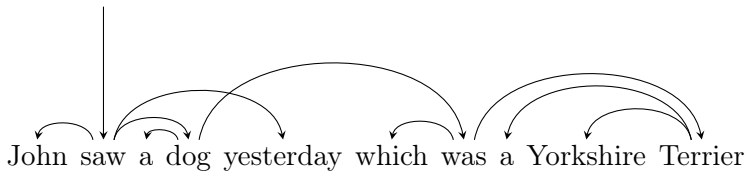

Figure 2.10: An example of a non-projective tree taken from the paper "Non-projective Dependency Parsing using Spanning Tree Algorithms" (McDonald et al., 2005).

Terrier" as shown in Figure 2.10. The relative clause in this tree "which was a Yorkshire Terrier" relates to the noun "dog", but the adverbial word "yesterday" is placed between and connected to the root. Using a projective structure, the relative clause could not relate to the noun because the arc shown going from "dog" to "was" had not been allowed to cross the line going from "saw" to "yesterday". This in turn will make for a less intuitive interpretation of the sentence, so that "was" either relates to "saw" or "yesterday".

For practical purposes, a projective parser is often preferred because they are in general faster, easier to implement and work with. This is often done even for languages like German where the theoretical framework wants non-projective structures. This is not always true, and there are parsers that support non-projective structures like the Maximum Spanning Tree algorithm suggested by McDonald et al. (2005). In this thesis however, we will use a projective structure to simplify the task of creating our corpus that will be described in Chapter 3.

2.2.3 Rule Based Systems

Rule based systems are systems that follow rules made by humans rather than learning from data. These systems were the dominant type of systems in the 80s and 90s. A big reason for this was because it came down to not having the machine power to analyze the required amount of data to make an efficient data-driven system. But they were also attractive because one could model a language formally, and the trees were closer to the linguistic theories.

Being that such systems are mostly written by human experts, they usually have a very high precision in parsing and gives trees that are linguistically informed and correct. If we were to write a parser for our toy grammar in Figure 2.6, and then instruct the parser to use the Context-Free Rules as a model for our language, it would have a rule based parser. This parser would only accept the sentences that we instruct it to.

The development of domain-independent hand-crafted grammars is a demanding enterprise because every rule in the language has to be had written and reviewed by a person. That means for it to be adopted to a new domain

```
(S
    (NP
      (D the)
      (N dog))
    (VP
        (VP (TV chased)
            (NP
                (D the)
                (N cat)))
        (PP (P around)
            (NP
                (D the)
                (N corner)))))
```

Figure 2.11: Sample Phrase Structure Tree written as a bracketed parse tree. The same tree that is shown in Figure 2.5

a big effort in making new rules will have to be made by qualified linguists knowledgeable in the syntactic framework. Rule Based systems are also often said have problems with robustness (Nivre, 2006) since they will not provide a parse for small errors in syntax or morphology, which is a problem in the context of speech because there are frequently errors, and a parser needs to handle them as well.

2.2.4 Data Driven Systems

The main idea behind data driven systems is to instead of having humans write rules for the parser, the machine should be able to teach itself the rules based on examples. This is done by having lots of example data, often referred to as treebanks in the field of Syntactic Parsing because they are collections of manually corrected parse trees for the sample sentences. The process includes elements of machine leaning, which will be introduced later in this chapter.

Figure 2.11 shows an example of a bracketed-style tree of the sentence "the dog chased the cat around the corner". The tree is exactly the same as the one in Figure 2.5 in this format. This bracketed style format is used to describe the trees found in the Penn Treebank, which is introduced in Section 2.3.2.

Using the one tree in Figure 2.11, we could extract a CFG grammar by walking through the tree and pick out the rules necessary to produce this tree. Our grammar would be exactly the same as the example found in Section 2.2.1 except that the list of non-terminals "E" would not contain the determiner "a", because that is not found in our training tree. If we had many such examples, the parser could learn many more rules and even the likelihood of which rules

are applied where and which tree is more likely as a whole than another. This is more commonly known as disambiguation. These kind of grammar systems are called Probabilistic Context-Free Grammars(PCFG).

The word *treebank* has been mentioned. A treebank is a large collection of annotated trees like the one shown in Figure 2.11 or the type seen in Figure 2.12 in the ConLL format. These large collections of annotated trees can be used to train parsers of different types depending on the data it contains.

An added effect of having large quantities of data to train and learn on, is that in the machine learning process the parsers can make generalizations. These generalizations can be applied to words or rules even if the parser has not seen a specific combination or instance of words. One such generalization we could have done regarding the missing "a" in our data-driven parser is that a determiner "D" is likely to precede a noun "N". Following this reasoning it is likely that given the sentence "The dog chased a cat." the unseen "a" is a determiner.

The treebanks takes a lot of effort by qualified people to make, but once you have a treebank the probabilistic systems are faster to make and adapt than rule-based systems. Probabilistic systems are easier to adapt to new domains by combining treebanks in more general domains and specific domains. Probabilistic systems are also more robust in that they can make trees out of anything, but instead assigns small probabilities to the trees that does not have relevant training data to back up the working hypothesis. In our case, we have a treebank of transcribed speech called the Swithboard treebank that is a part of the Penn Treebank which will be introduced later.

Data-driven dependency parsing

The Maltparser is a data-driven transition-based dependency parser (Hall, 2008) and a collection of different data-driven dependency grammar algorithms both Projective and Non-Projective. It uses, among others, Support Vector Machines (see Section 2.5) to train a parse-guide.

The Malt Parser has proven to be highly flexible both in differences in languages (Nivre et al., 2007) and domains (Nivre, 2007). The Maltparser should therefor be well suited for the task of parsing spoken language.

Data format

ConLL is the format that we will be using for the Dependency Grammar treebanks. It is a broadly adopted format that is used by e.g. the Maltparser that we will be using. Figure 2.12 shows the ConLL version of the sentence "The dog chased the cat around the corner" and represents the same tree as the one shown in Figure 2.9.

The ConLL format is given as a tab separated feature list, where one token is on one line each. Each token can have ten or more features depending

ID	FORM	LEMMA	CPOS	POS	FEATS	HEAD	REL
1	The	_	D	D	_	2	DET
2	dog	_	N	N	_	3	NSUBJ
3	chased	_	V	V	_	0	ROOT
4	the	_	D	D	_	5	DET
5	cat	_	N	N	_	3	DOBJ
6	around	_	P	P	_	3	PREP
7	the	_	D	D	_	8	DET
8	corner	_	N	N	_	6	POBJ

Figure 2.12: The sentence found in Figure 2.9 written in the ConLL format.

on the language and data set requirements. We only display eight of them in Figure 2.12 because we do not use the last two (these are "PHEAD" and "PREL"). If there is no applicable or available data for the current word and feature, the "_" is placed in its stead.

Not all the features are available in our data set, but a brief description of all the features will be provided. The bracketed features in the list are the one that we do not have.

- *ID:* A numeric value to show where in the sentence a word-token is.
- *FORM:* The surface-level form of the word-token.
- *[LEMMA]:* The lemma or the stem of the word-token.
- *CPOS:* Short for Coarse-grained Part-Of-Speech Tags, and contains less fine-grained Parts-Of-Speech than the "POS" column.
- *POS:* The Part-Of-Speech tags for a given word-token, with more specific tags than "CPOS" if available. In our data set this feature contains the same value as the previous column "CPOS".
- *[FEATS]:* A list of syntactic or morphological features for a given token.
- *HEAD:* shows which token is the head of this token using the "ID". "root" is given the value 0.
- *REL:* The Relation variable gives us the arc label or the function name of the connection between the dependent and its head.
- *[PHEAD & PREL]:* The "P" in both variables stands for "Projective" and, if present, gives a projective version of the sentence in question.

2.3 Spoken Language

Parsing natural language in its written form is a big topic in itself, but when it comes to spoken language, some extra challenges arise. The extra challenges come from the more informal and real-time nature of dialogs. Participants may not have the time necessary to formulate a complete sentence and starts saying something. They might realize some time later that they started to say the wrong thing and have to correct themselves or having problems completing the sentence later on.

This section will outline some of the issues found in spoken language as opposed to written language. We will see how these characteristics are annotated in the Penn Switchboard corpus and the motivation behind building a treebank that incorporates the phenomena found in spoken language.

It is important to note that when we are dealing with spoken language, the term *sentence* should be distinguished from the term *utterance*. This is because an utterance often roughly corresponds to what we know as a sentence, but may be incomplete or structured in different ways to what we normally think of as complete sentence. Because output from an ASR component often do not contain punctuation, it also becomes a little harder to talk about sentences rather than a collection of words representing the speakers intent.

2.3.1 Phenomena in Spoken language

The following list gives an overview of the different phenomena that exists in spoken dialog that do not occur in written form.

- *Repairs:* When people are trying to express themselves, they often make a mistake such as choosing the wrong word, changing their mind on what they wanted to say or simply stopping for time in order to figure out the next word. This often comes out in a dialog as a disfluency. What happens is that the person talking is changing what he wants to express and then abruptly ends the current line of thought and starts a new one. e.g "I, we can't think" contains a change where the user exchanges the noun "I" with "We". If it had been written, this person most likely would have stopped up before writing anything and thought about what he wanted to write and written it in a more syntactically correct way according to the rules of written language instead of saying it again.

- *Duplications:* A phenomena similar to repairs is duplications. Duplications happens in much the same way as repairs, only instead of changing the utterance, it is confirmed. If the speaker of the sentence we saw in the "Repairs" section had said "I" again instead of "we" it would be an example of a duplication.

- *Deletions:* Another form of dialog disfluencies are deletions. These happens when the speaker changes their mind about the entire phrase, and instead of repairing the utterance the speaker indicates that the user should instead forget what was said previously and makes a new phrase. e.g "The Wall, um, How many albums did pink floyd make?", where "The Wall" is the start of a dropped phrase that the speaker did not finish.

- *Meta communicative dialog acts:* Another thing people often do to stall for time so they can think about what they want to say, is showing that they are thinking or are not finished by using certain utterances related to their language. This manifests itself mainly in two distinct forms. Saying something that indicates you are still in the process of saying something like "well" in e.g "well, maybe it wasn't that one.", or dragging out words like "um" in e.g. "The bands name was, um, Led Zeppelin" where the speaker "ums" in order to indicate that he is trying to recall the artists name.

- *Fragmentary utterances:* A dialog requires at least two people, and people will often utter the shortest phrase possible in order to convey their meaning. This often leads to utterances in the dialogs which are not complete sentences but just the parts of them that the listener needs to hear in order to understand what the speaker intended to convey. As such, the listener may also interrupt the speaker before he is done in order to show that he thinks he has understood what the speaker tried to convey. The interpretation of such non-sentential utterances has notably been studied by Fernández (2006).

- *Contextual factors:* When people are speaking to each other in person or via video chat people can see each other. Talking in this manner, they often use gestures and similar in order to convey their meaning. This in turn makes the listener able to complete the conveyed message even though the speaker may never complete or indeed say anything at all.

  In the context of this paper this would be hard to do anything about. This is because the corpus, the Penn Switchboard Treebank (introduced in the next section), we are dealing with phone conversations and this does not occur. This is also something a syntactic parser can help with without external information. But it is a problem one should be aware of because it is a hinder to finding out the semantic meaning of a conversation. e.g A: "look outside." where B replies "Yeah.", where we would have to be able to look at what they are looking in order to know the meaning.

- *Grounding:* A phenomenon that allows speakers to confirm that a utterance was received and understood. This process allows the participants

```
( (S
    (NP–SBJ (PRP I) )
    (, ,)
    (INTJ (UH uh) )
    (, ,)
    (VP (VBP listen)
      (PP (IN to)
        (NP (PRP it) ))
      (NP–TMP (PDT all) (DT the) (NN time) )
      (EDITED
        (RM (–DFL– \[) )
        (PP–LOC–UNF (IN in) )
        (, ,)
        (IP (–DFL– \+) ))
      (PP–LOC (IN in)
        (RS (–DFL– \]) )
        (NP (PRP$ my) (NN car) )))
    (, ,) (–DFL– E_S) ))
```

Figure 2.13: A tree taken out of the Switchboard Corpus. The original utterance was "I, uh, listen to it all the time in, in my car,".

> in a conversation to achieve mutual understanding. This is most commonly done implicitly by the listener by using parts of the utterance in a reply to the speaker. It can also be done explicitly by using affirmative type statements like "yes" and "no" (Traum, 1991; Traum & Allen, 1992).

These phenomena have to be handled by a dialog system. The task of the NLU component is to deal with a lot of these problems and to build a correct representation of an utterance. We will try to address some of them using syntactic parsing.

2.3.2 Penn Switchboard treebank

The Penn Switchboard Treebank is a large collection of bracketed Constituent Grammar syntax trees similar to the one found in Figure 2.11. Together with the Penn ATIS Treebank, it is one of the biggest treebanks for spoken language.

The Switchboard corpus consists of transcribed conversations that took place over the phone between people. The ATIS corpus is a treebank of transcribed interactions with a automated flight ordering system called ATIS. In this thesis we want to have spoken language that flows in the same manner as between humans, and for that reason we will focus on the Switchboard Treebank.

In addition to following the Penn-style annotation for the treebank, the Switchboard Treebank contains extra annotation to facilitate some of the problems described in the previous section. They specifically identify the Repairs and Deletions, Incomplete utterances and Meta communicative dialog acts. Our example tree in figure 2.13 shows an example of all three phenomena. We will look at them in turn in the following sections.

Repairs & Deletions

Repairs and deletions constitute the most notable difference in the Switchboard trees compared to the written portions of the Penn Treebank, and they are annotated in the surface form as well as in the parse trees of the utterances.

When talking about Repairs and Deletions, there are three things we will talk about. The *restart* which is the whole repair, duplication or deletion; The *reparandum* which is the part of the restart that is removed; And the *repair* which is the part of the restart that replaces the reparandum in the utterance.

The annotation in the Switchboard corpus for capturing the repairs are brackets around the entire restart. The reparandum and the repair is also separated by a marker. (Meteer, Taylor, MacIntyre, & Iyer, 1995) The annotation uses the following three character sequences to annotate this:

- \/ Marks the start of the restart and the beginning of the reparandum.
- \+ Marks the end of the reparandum and the start of the repair.
- \/ Marks the end of the repair and the restart.

We can see this annotation in our example tree in figure 2.13. If the utterance in that tree was to be written with the restart symbols, it would look like this: "I, uh, listen to it all the time [in, + in] my car,". The reparandum would be "in," before the + marker, and the repair would be "in".

Deletions are annotated in a similar manner, only without a repair. An example of this would be "\[The Wall , \+ \] um, How many albums did Pink Floyd make?" where the phrase "The Wall" is marked for deletion.

Incomplete Words & Utterances

People are sometimes stop in the middle of utterances or words either because they are interrupted by another speaker, finished what they have to say before it is a complete sentence or want to change their utterance. Then we have an incomplete word or utterance.

In the Switchboard corpus, this is shown by adding a N_S or a E_S tag to the end of the utterance. The N_S and E_S represent incomplete and complete sentences respectively. For the trees, the annotation is treated the same way as punctuation placing them as close to the root as possible. We

can see this in our example tree in figure 2.13 in the very last line. There may be more than one such tag in a parse tree and it shows what the annotators thinks constituted a complete utterance.

When the interruption happens in words, the change is annotated on the surface level and in the part of speech tags. The start of the incomplete word is written with a - on to show that the word stopped here. The corresponding part of speech tag for that word is set to "XX". E.g "Why would you wa- . N_S" where the speaker meant to ask "why do you want that?", but decided to cut it short.

Meta communicative dialog acts

The meta communicative dialog acts are also available in the Switchboard Treebank. The Constituent trees has the "(INTJ ..)" clause to mark these acts. Figure 2.13 shows how this looks like in the Switchboard Treebank with "uh".

In addition to the "INTJ" structure in the tree, the "uh" also has a special POS tag "UH", which helps us identify the word-tokens that the meta communicative dialog acts consists of.

2.3.3 Previous Work

Parsing spoken language has been an active area of research for the last two decades. Here we will look at some of the approaches and results that has been made, especially in the context of the Switchboard Treebank.

One of the early attempts of parsing the Penn Switchboard Treebanks were performed by Charniak and Johnson (2001). The approach that they took was to use a edited-word detector and Wallstreet Journal Phrase-Structure parser, i.e. a parser trained on the Wallstreet Journal treebank of written text. The detector were designed to find all the restarts and remove the reparandum. The Wallstreet-Journal parser was then used to parse the resulting sentence.

The detector achieved 95% precision and 68% recall in finding the restarts using the gold tags. The parser managed to do a Precision and Recall on the resulting sentences of 85.4% and 86.6% respecively. With gold edits, the parser managed to do Precision/Recall on 87.8% and 88.1%.

Another study done by Jørgensen (2007), examined the effects of the disfluencies found in the Switchboard corpus on two parsers, Dan Bikel's Parser and the Maltparser. The study shows that the data-driven Dependency parsers increase in accuracy when disfluencies are removed.

The last paper we want to note is the preliminary work done by Joakim Nivre on the Swedish Spoken Language Treebank (Talbanken05), using the Maltparser (Nivre, 2007). This paper compares the difference of the Labeled and Unlabeled Accuarcy score for a written and a spoken treebank. The paper shows that the parser drops 6-7 percentage points for Projective parsing and

2-3 percentage points with Non-Projective parsing. It concludes that there is reason to believe the gap can be closed by a better adaptation of the syntactic representation in the spoken treebank.

2.4 Dialog Acts

To make dialog managers and core components of the NLU and Natural Language Generation more manageable and reusable a description of human dialogs is practical. Dialog acts serves as a formal representation of the semantic and pragmatic function associated with particular utterances (Austin, 1975; Searle, 1976).

For the context of this thesis we will use the same definition of a "dialog act" as the one found in Bunt (1995).

> "We define the notion 'dialog acts' as the functional units used by the speaker to change the context."

To make the high level description of a conversation, a unit that is independent of the utterance is necessary. This is because utterances may contain more than one dialog act. Take the utterance we took from the Switchboard Treebank in Figure 2.13.

Using the entire utterance "I, uh, listen to it all the time in my car", the speaker is informing the listener about the fact that the speaker listens to something all the time (a statement). But you might also say that the "uh" is information to the listener that the speaker intends to continue speaking (an interjection).

Dialog acts incorporate different types of information. An overview of the most common elements encoded in dialog acts will be described in this section. We do not give a complete overview of the amount of information that can be incorporated into dialog acts, but it should give the reader a sufficient overview and an idea of what information might be incorporated into a dialog act.

2.4.1 Conversation structure

A conversation between humans is a social and joint activity done by two or more speakers. Because it is a joint activity, conversations are structured. This structure is evident in the turn-taking nature of a conversation where each speaker will wait for his or her turn to speak before starting an utterance.

The turn taking rules as described by Sacks, Schegloff, and Jefferson (1974) are the following:

i) If the speaker assigns A as the next speaker, A must take the next turn.

ii) If the speaker assigns no-one to be the next speaker, anyone can take the next turn.

iii) If no-one takes the next turn, the speaker may take the turn again.

These rules only applies at transition-relevance places, the place where the conversation allows for changing speaker. This transition marks the end of an utterance from one speaker, regardless of whether the utterance is a complete sentence or syntactic unit.

Since conversations are joint, the agents in the conversation have to have some common ground on which to build the conversation. This is called grounding and happens all the time in conversations. A big source of grounding is the constant feedback a listener gives in terms of "uh-huh" and "mm" signifying that the listener is receiving what the speaker is saying. When this feedback happens in utterances it is called an act of grounding and was described inSection 2.3.

2.4.2 Speech Acts

The idea of utterances as acts was conceived by Wittgenstein in 1953, but Austin formalized it at a later point(Austin, 1975). Austin claimed that all utterances in a real speech situation could be decomposed into three dimensions of what he called speech acts: locutionary, illocutionary and perlocutionary acts (the term speech act is also often used to mean just the illocutionary acts). They describe the following three categories of utterances:

- *locutionary* signifying a special meaning.
- *illocutionary* asking, commanding, answering etc.
- *perlocutionary* causing the addressee of the utterance to change psychical state. e.g. scaring.

The illocutionary acts where further broken down into 5 types of acts made by Searle (1976).

- *Assertives:* asserting that something is true. Boasting, suggesting, concluding ...
- *Directives:* Directing the addressee to do or reply something. Advising, ordering, asking ...
- *Commissives:* A promise of something being done in the future. Opposing, promising, panning ...
- *Expressives:* Describing the attitude or emotions the speaker has about a state of affairs. thanking, deploring, apologizing ...
- *Declarations:* Changing the state of the world by an utterance. E.g "I quit the job!"

Dialog acts are often looked at as an extended form of a speech act. This means that while they sometimes mean the same they are often used to describe speech acts plus information from other sources like dialog context or conversation structure as described in the previous section. The variations in how dialog acts are described is reflected the way the tag sets for dialog acts are described. The next section will look at the two major tag sets for the Switchboard corpus.

2.4.3 DAMSL & NXT Tag Set

Because there are so many things that can be encoded into dialog acts, tag sets differ in what they encode and how they encode it. We will touch on two that have been used in combination with the Switchboard corpus, Dialog Act Markup in Several Layers (DAMSL) and NXT.

The DAMSL tag set annotates three types of information in its tag set (Core & Allen, 1997). The tree layers are called the "Forwards Communicative Function", the "Backwards Communicative Function" and "Utterance Features".

- *The Forwards Communicative Function* deals with about the same taxonomy as the one found in the previous section on speech acts.
- *The Backwards Communicative Function* tells us how this utterance relates to the previous one.
- *Utterance Features* shows information about the utterances form and content. I.e is the utterance uninterpretable.

This tag set, with some additions was, applied to the Penn Switchboard corpus. The tag set was then clustered into 42 categories (SWBD-DAMSL) based on that a lot of categories had few or no utterances. (Jurafsky, Shriberg, & Biasca, 1997)

The NXT tag set is a glossing for the SWBD-DAMSL tags, and the tags in the tag set are named a little more intuitively. Table 2.1 shows a description of the 42 tags used in the NXT dialog act tag set. The table is taken out of The NXT-format Switchboard Corpus (Calhoun et al., 2010). The article also provides a mapping from the NXT Tags to the SWBD-DAMSL tags, for the interested reader. In this thesis we use the NXT glossings for testing and classification.

2.4.4 Previous Work

The field of automatic dialog act tagging has been explored before. Most notably a similar system using an extended form of context-free grammars called Definite Clause Grammars (Pereira & Warren, 1980) was proposed by Van Noord, Bouma, Koeling, and Nederhof (1999). This paper concludes that

a grammatical analysis of user utterances is fast enough and effective for use in the task of Spoken Language Understanding.

Stolcke et al. (2000) present an extensive approach to dialog act classification using a wide range of lexical, collocational, contextual and prosodic features. Their approach is evaluated on the Switchboard corpus and results in a dialog act classification accuracy of 65% when applied to automatically recognized words. They also do experiments using transcriptions where they achieve an accuracy in classification of 71%.

Other papers on the topic include Araki (2010) that uses cue phrases to classify the Switchboard Corpus and SWBD-DAMSL. They achieve an classification accuracy of 57.1%. And the thesis of Webb (2010), also using cue phrases on the Switchboard corpus, achieving an accuracy of 69.65%.

2.5 Machine Learning

In this thesis we use Machine Learning (ML) when we train our Data-Driven Dependency parser, and we use Machine Learning when we classify the Dialog Acts. We should therefore know a little bit about the topic.

For a machine to learn patterns from examples, we would have to design algorithms that enables the machine to do so. If we could do that, the machine could learn to do all sorts of tasks itself. This approach to solving problems has been very successful in a variety of tasks, including Natural Language Parsing. Most of the modern Dependency-Grammar parsers use Machine Learning in some variant.

2.5.1 Definitions

The goal of Machine Learning can best be explained in terms of "Well-Posed Learning Problems". This term is taken from Tom M. Mitchells book on Machine Learning (Mitchell, 1997), and it is defined as the following:

> "A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performace at tasks in T, as measured by P, improves with experience E"

If we where to propose such a learning problem for our dependency parsers, it could look something like

- Task T: Parse sentences in Natural Language
- Performance measure P: Percentage of Dependency Relations correctly attached and labeled.
- Training experience E: Correctly parsed sentences given by humans.

In this context, E is the Treebanks that we have talked about in Section 2.3 and P is the "Labeled Attachment Score" and "Unlabeled Attachment Score".

Labeled & Unlabeled Attachment Score

Labeled Attachment Score (LAS) and Unlabeled Attachment Score (UAS) are the two main evaluation metrics that we use for Dependency Parsing. They measure the following characteristics of the dependency trees.

UAS: The number of tokens that is attached to the correct head divided by the total number of tokens. This corresponds to the number of correct IDs in the "HEAD" column of the ConLL format.

LAS: The number of tokens that has both the correct head and the correct function assigned to it divided by the total number of tokens. In terms of the ConLL format, this corresponds to having the correct ID in the "HEAD" column, and the correct function label in "REL".

2.5.2 Implementations

A large number of machine learning algorithms have been developed, and they all have their pros and cons. In our thesis, Support Vector Machines (SVM) is the machine learning algorithm that we use in both the Dependency-Grammar parser and the classification.

Support Vector Machine was introduced by Cortes and Vapnik (1995) and is one of the most popular algorithms in machine learning today (Marsland, 2009). SVM is a maximum margin classifier. A maximum margin classifier is a classifier that finds a optimal straight line through a data set by maximizing the margins between two classes and separates them by setting the line in the middle of the margin.

To separate support vectors that can not be separated in this way in this linear fashion, a SVM can map the vectors to a higher plane. This is done by using a kernel function. The kernel function can be applied infinitely or until the SVM finds its optimum. For data sets with long support vectors this operation is time consuming, and not always necessary. Natural Language tasks often fall into this category.

We rely on the support vector machine implementations called libSVM and liblinear, written by Chang and Lin (2011).

2.6 Summary

In the background chapter we examined the fundamental concepts that we are going to use and take advantage of in our work.

Dialog systems are systems designed to listen and respond to a user using speech.

Phrase structure grammar and *dependency grammar* are both syntactic frameworks for describing natural language and how sentences are built up. The fundamental difference between them is that Phrase Structure Grammar builds trees containing Phrase Structures, and Dependency Grammars builds trees of word-to-word relations. *Arc labels* in Dependency-Grammar tell us the syntactic function of a word.

Data-driven approaches are usually more robust than *rule-driven* approaches.

Spoken language is not the same as written language and contains extra problems for a natural language parser, like restarts and deletions. There has been done some work to mitigate this, mostly by removing the extra problems before the parsing (Charniak & Johnson, 2001; Jørgensen, 2007), but also promising results by using data-driven dependency-parsers directly (Nivre, 2007).

The Switchboard corpus contains annotated data for some speech phenomena. These phenomena include repairs, duplications, deletions and hesitations.

Dialog acts is a way of representing the pragmatic intention that underlies the users utterance, ranging from grounding to questions.

We have also presented data sets, formats and tools.

- *Penn treebank* – A Phrase Structure Grammar Treebank containing annotated data from the Wallstreet Journal and Switchboard conversations.
- *ConLL* – An annotation format for dependency graphs.
- *NXT & DAMSL* – Two different dialog act schemes, where we chose to use the NXT style tags.
- *Maltparser* – A data-driven dependency parser using SVMs which contains a number of different dependency parsing algorithms.
- *libSVM & liblinear* – The two implementations of Support Vector Machines that is used in the Maltparser, and we will use in our classification experiments.

NXT Tags	Summary	Example
abandon	Adandoned or Turn-Exit	So, -/
acknowledge	Response Acknowledgment	Oh, okay.
affirm	Affirmative non-yes answers	It is.
agree	Agree/Accept	That's exactly it.
ans_dispref	Dispreferred answers	Well, not so much that.
answer	Other answers	I don't know.
apology	Apology	I'm sorry.
apprec	Appreciation	I can imagine.
backchannel	Backchannel	Uh-huh.
backchannel_q	Backchannel as question	Is that right?
close	Conventional-closing	It was nice talking to you.
commit	Offers, Options & Commits	I'll have to check that out.
completion	Collaborative Completion	or not.
decl	Declarative Wh-Question	You are what kind of buff?
directive	Action-directive	Why don't you go first
downplay	Downplayer	That's all right.
excluded	Excluded - bad segmentation	-
hedge	Hedge	Well, I don't know.
hold	Hold before response	I'm drawing a blank.
maybe	Maybe/Accept-part	Something like that.
neg	Negative non-no answers	Uh, not a whole lot.
no	No answers	No.
open	Conventional-opening	How are you?
open_q	Open-Question	How about you?
opinion	Statement-opinion	I think it's great.
or	Or-Clause	or is it more of a company?
other	Other	I tell you what.
quote	Quotation	[I said] "Okay, fine"
reject	Reject	Well, no.
repeat	Repeat-phrase	Oh, fajitas.
repeat_q	Signal-non-understanding	Excuse me?
rhet_q	Rhetorical-Questions	Who has time?
self_talk	Self-Talk	What is his name?
statement	Statement-non-opinion	He's about five months old.
sum	Summarize/Reformulate	So you travel a lot.
tag_q	Tag-Question	Right?
thank	Thanking	Hey thanks a lot.
third_pty	3rd-party-talk	Katy, I'm on the phone.
uninterp	Uninterpretable	But, uh, yeah.
wh_q	Wh-Question	Well, how old are you?
yes	Yes answers	Yes.
yn_decl_q	Declarative Yes-No-Question	You just needed a majority?
yn_q	Yes-No-Question	Is that what you do?

Table 2.1: Description of the 43 tags that are used for dialog act classification in NXT. The table is taken from the NXT documentation.

Chapter 3

Dependency Parsing of Spoken Language

3.1 Motivation

The selection of a parsing paradigm for natural language is in a large sense an exercise in trade-offs. An optimal system would be accurate, robust, fast and provide as much syntactic and semantic information as possible. In the real world, systems often sacrifice one of these aspects in order to gain an advantage in an other aspect of the parser and choosing a system comes down to the task at hand.

In this thesis we are investigating the effect syntactic parsing can have as a part in a spoken language understanding component. More specifically, we investigate whether we can improve dialog act recognition by using the information provided by dependency parsers.

We want a parser that can always deliver a dependency tree for any utterance given. We remember from Section 2.3 utterances do not always conform to strict syntax rules. This means that we need a robust parser that is able to provide enough information to help the dialog act recognition

The dependency parsing experiments by Nivre that we looked at in Section 2.3 show that data-driven dependency parsing is a good candidate for parsing systems for spoken language understanding. For those reasons we want to investigate the possibility of augmenting the Switchboard Treebank to better suit a dependency parser for spoken language.

Spoken language is diverse and much more irregular than written language, it is important for any parser that is going to attempt to deal with it to be very robust. It is also preferable that such a parser is also capable of parsing incomplete syntactical units so that even utterances that either are interrupted or rely on common knowledge to be completed, gets a syntactic analysis. A dependency grammar approach would be able to handle this kind of incompleteness since it models only relations between words and a tree can

be made by connecting the existing words instead of waiting for a complete phrase (Nivre, 2005).

Speed and linguistic complexity are often in conflict in the trade-off problem. This is no different in the domain of natural language processing, so it is important to try to have a system that is fast enough and still delivers information that can help in the act recognition. Dependency trees provide a sparser representation than those found in phrase structure grammars and can therefore perform faster. The dependency-based representations are also makes the representation closer to the semantic relationships of the words, which makes them more directly usable for reading the syntactic information to use in a dialog act recognition situation.

However, the amount of training data available for data-driven dependency parsing of spoken language is not large. For English, the main resources of annotated semantic and syntactic data are the Penn Treebanks. These are phrase structure trees and not dependency trees, and they will have to be converted to be used to train a dependency parser. While the written parts of the corpus, Wall Street Journal and Brown, has "off-the-shelf" phrase structure to dependency converters written by Johansson and Nugues (2007) and De Marneffe, MacCartney, Manning, et al. (2006), there is no such converter for the Switchboard part of the corpus. Such a treebank is needed to train a data-driven dependency parser and it should be in the domain of spoken language to train the parser on phenomena that occur in speech.

The main goal of this chapter is to describe the creation of such a treebank from the output of one of the "off-the-shelf" phrase structure to dependency conversion tools written for the Penn Treebanks. Since the converters are not made for the switchboard corpus, the creation has to involve the conversion of the necessary structures and labels to make it into a proper dependency treebank specifically for spoken language. This include augmenting the dependency label set in order to capture some of the phenomena mentioned in Section 2.3 on Spoken Language. In the following we will describe our conversion algorithm in detail. We will then go on to present a set of parsing experiments where we train a data-driven dependency parser on the converted treebank and compare it with dependency parsers trained on other versions of the treebanks.

3.2 Converting From Phrase Structure to Dependency Representation

Before we are able to use the Penn Switchboard corpus to train a dependency-based parser, it is necessary to convert it from the phrase-structure form found in the Penn Treebanks to a dependency format. Figure 3.1 is a tree taken from the Penn Switchboard corpus. This tree has the same bracketed annotation described in Section 2.2.4.

```
(S
(EDITED
  (RM (-DFL- \[) )
  (NP-SBJ (PRP I) )
  (, ,)
  (IP (-DFL- \+) ))
(CC but)
(NP-SBJ (PRP I) )
(RS (-DFL- \]) )
(VP (VBD was)
  (PP-PRD (IN from)
    (NP (JJ conservative) (NNP Wisconsin) )))
(, ,) (-DFL- E_S) )
```

Figure 3.1: A sentence taken from the Penn Switchboard corpus.

We looked at two different converters that convert the Penn-style phrase-structure trees to a dependency representation. The first one is the Pennconverter (Johansson & Nugues, 2007) from Lund university, and the second one is the converter that is part of the Stanford Parser (De Marneffe et al., 2006) package, which will be referenced as the Stanford Converter.

Looking at the tree in Figure 3.1 you might notice the sub-tree "EDITED". This sub-tree contains the disfluency annotation discussed in Section 2.3. The Switchboard trees and the extra annotation needs to be converted into a dependency format for us to use it to train a Data-Driven Dependency parser. We can convert them in 2 ways. Either remove all the edits that we talked about, or keep them and let the dependency parser acquire this extra information.

The first approach would remove the disfuencies entirely and treat the Switchboard treebank as if it was written text. To parse new text, we would have to pre-process the input to remove the edits using an edit-detection algorithm similar to the one done by Charniak and Johnson (2001).

Since we want to look at how syntactic parsing might aid the task of dialog act recognition, it seems preferable to retain as much syntactic information as possible. Therefore we want to conserve as much of the original data found in the Switchboard about the real nature of spoken language. For the initial conversion we investigate the use of two parsers designed to convert the Wall-street Journal and Brown Treebanks of written text from phrase structure to dependency trees.

3.2.1 Initial Conversion

To do the initial conversion we tested the Pennconverter (Johansson & Nugues, 2007) from Lund University and the Stanford converter (De Marneffe & Man-

Converter	Conversations	Sentences	Tokens
Pennconverter	650	90127	728359
Stanford converter	650	110504	1233722

Table 3.1: An overview of the resulting sentences and words in the different converters.

1	\[	_	-DFL-	-DFL-	_	8	dep	_	_
2	I	_	PRP	PRP	_	1	dep	_	_
3	,	_	,	,	_	1	punct	_	_
4	\+	_	-DFL-	-DFL-	_	1	dep	_	_
5	but	_	CC	CC	_	8	cc	_	_
6	I	_	PRP	PRP	_	8	conj	_	_
7	\]	_	-DFL-	-DFL-	_	8	dep	_	_
8	was	_	VBD	VBD	_	0	root	_	_
9	from	_	IN	IN	_	8	prep	_	_
10	conservative	_	JJ	JJ	_	11	amod	_	_
11	Wisconsin	_	NNP	NNP	_	9	pobj	_	_
12	,	_	,	,	_	8	punct	_	_
13	E_S	_	-DFL-	-DFL-	_	8	dep	_	_

Figure 3.2: A sentence taken from the Stanford conversion of the Switchboard.

ning, 2008), which is a part of the Stanford Parser package.

It is worth to note that neither converters were designed to use on the Switchboard part of the Penn Treebanks, which means that none of them support any of the disfluency-type notation found in the Switchboard trees. This makes it interesting to see what happens when we try to apply the different converters on the Switchboard corpus.

Overview

Table 3.1 shows the number of sentences and tokens in the output of the Pennconverter and the Stanford converter when we apply them to the Switchboard Corpus. The outputs of the two converters differ by quite a lot both in terms of sentences and word-tokens. The converters handles the data in different ways, and we will have to select one that produce something we can work with.

Stanford Converter

The Stanford Converter was not able to handle the special constructions that were found in the disfluency annotation of the Switchboard corpus, and couldn't assign a label to the special words that we find in spoken language.

As a result, we got a lot of "dep" relations to words with POS-tags which are not in the Wallstreet Journal and the disfluency markings \ [, \ + and \ ].

The unknown disfluency tags were retained in the resulting dependency trees. This resulted in dependency trees of the type we see in Figure 3.2 written in the ConLL format that we introduced in Section 2.2.4. The retention of the tags makes it possible to use the disfluency information.

If we want to do a edit-detector approach, we could introduce these tags prior to parsing. That would enable us to use a parser trained on the trees as they appear in the Stanford converter output could be used. But if we want to approach it with the intent of making the Maltparser process these phenomena, the treebank would have to be post-process to make a dependency tree with the disfluency tags incorporated into the function labels rather than surface forms.

the *dep* function In the tree in Figure 3.3 we can see the restart symbols and they are connected using the dep function. The dep function is used in the stanford label set as the root of the label hierarchy that the Stanford Converter uses when converting the Phrase Structure trees. Having dep relations is undesirable because it gives no syntactic information outside that the two words are related and is roughly the same as having no tag.

Penn Converter

The Penncoverter also has problems dealing with the disfluencies that are found in the Switchboard Treebank. The trees that the Pennconverter is not able to convert in the Switchboard Treebank is dropped. This can be seen by the amount of sentences missing from its output in Table 3.1. This is also evident in the fact that the number of sentences containing the restart, as is shown in Table 3.2, roughly correspond to the number of missing sentences from the Pennconverter output.

Result

The Stanford Converter was chosen to do the initial conversion as it converted all the trees in the treebank. The fact that the restarts was also retained in the trees from the Stanford Converter enabled us to use them in a post-processing step and keep the disfluencies found in the Switchboard Treebank.

3.2.2 Disfluencies in the Converter output

For the task of dialogue classifications we want to keep information on the restarts and disfluencies found in the Switchboard corpus in the trees. Restarts could be interesting for a classification system as they show hesitation and uncertainties from the speakers side. In addition, if the parser was trained

to handle the disfluencies and restarts, we would not need to pre-process the input.

Contrary to Charniak and Johnson (2001) approach, our approach retains the syntactic information encoded in the restarts and does not require an additional parsing step.

In order to convert restarts and disfluencies, we introduce two new dependency functions "repair" and "hesitation". The parser should then be able to handle the disfluencies itself, without training a edit-detection component to remove the problematic parts. This section will describe a range of disfluency phenomena that occur in the converted Switchboard trees and the issues related to the speech annotation in the converted Switchboard Treebank.

Restarts & Duplicates

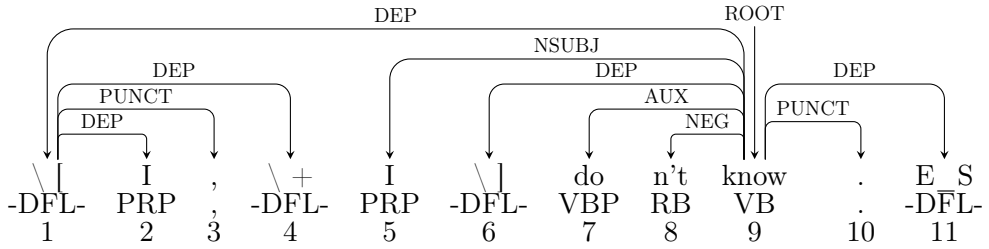

Figure 3.3: A tree that is taken directly from the Stanford converter output and shows the base case for a repair.

The sentence shown in Figure 3.3 is a typical example of how a complete sentence from the Stanford converter looks after it has been converted from its constituent form to its dependency form. Straight away we can see some parts of the surface form which does not belong in a pure transcript of the sentence, namely word 1, 4, 6 and 11 , but are there because they were introduced as disfluency annotations in the Switchboard corpus.

They are still there because the Stanford Converter was unable to handle them. In a dependency tree they would ideally be properly labeled with a function that corresponds to the restart tokens and in that way incorporate the corresponding relations without the extra word-tokens. This will be the main role of the repair function that is introduced later in this section.

Deletions

The tree in Figure 3.9 is an example of how a deletion is represented in the Stanford Converter output. Word number 2, "The", is the word that is in the deleted position. The deletion cases are also marked as repairs in our processing of the treebank.

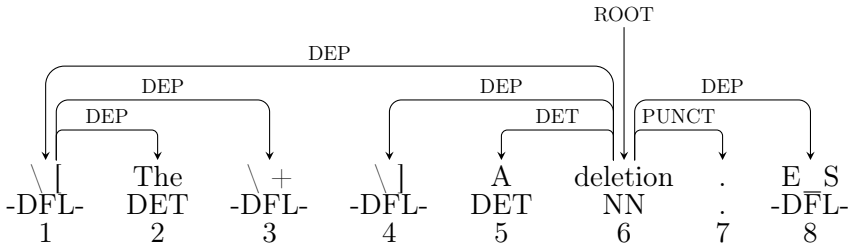

Figure 3.4: A deletion as seen in the Switchboard corpus.

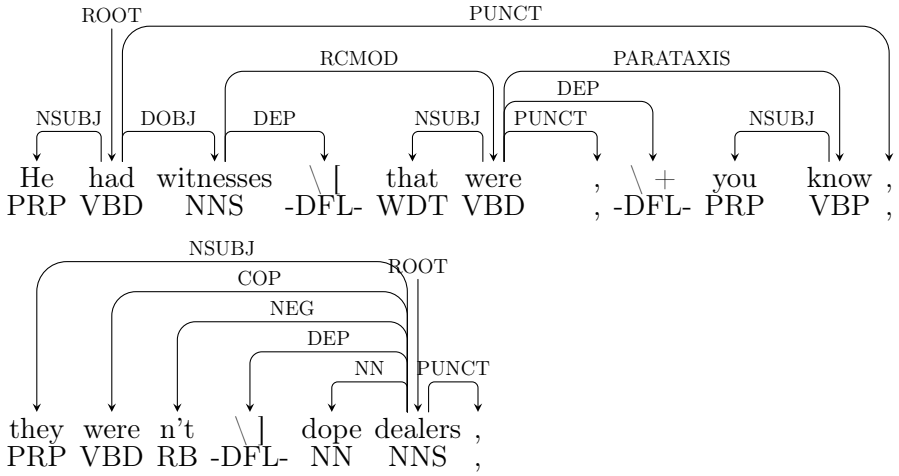

Figure 3.5: Two sentences with different types of unbalanced brackets.

Uneven Brackets

Another problem that is common in the output from the Stanford converter is that the utterances are split into pieces. This happens where the speakers change turns or give implicit feedback like backchannel grounding leaving the repairs spanning multiple utterances. Two such examples are shown in figure 3.5.

The utterances in figure 3.5 are the ones that correspond to each other, meaning that the utterances are part of the same statement "He had witnesses that where, you know, they weren't drug dealers". But they are split in the switchboard corpus for the reasons mentioned in the last paragraph. This means that the restart brackets are only partially present in each of the utterances and needs some special case compared to the complete restarts.

Nested Brackets

An utterance that has been repaired could itself be a repair of a previous utterance. Figure 3.6 is an example taken from the output of the Stanford

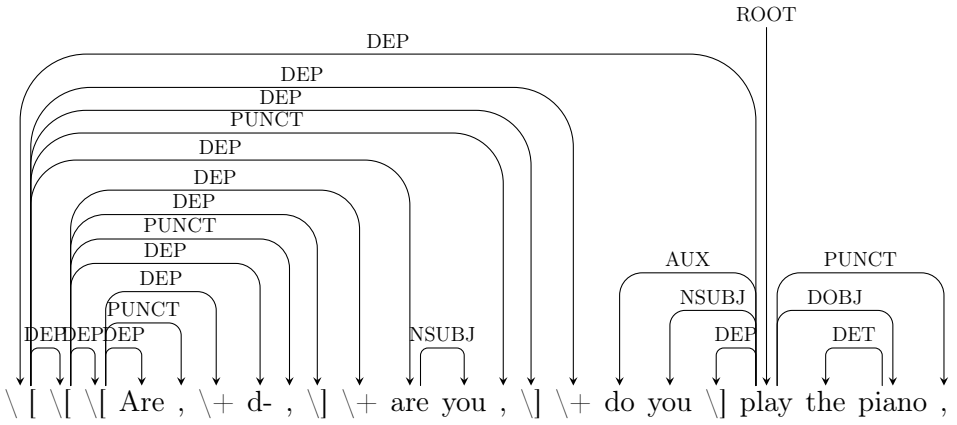

Figure 3.6: A unprocessed dependency tree containing nested repairs.

Converter containing three repairs nested. This utterance can in turn contain even further repairs or deletions. The recursive pattern that emerges from this structure is shown in the Switchboard corpus as nested brackets and has to be handled individually and unraveled in a way so that the structure is still preserved.

Hesitations

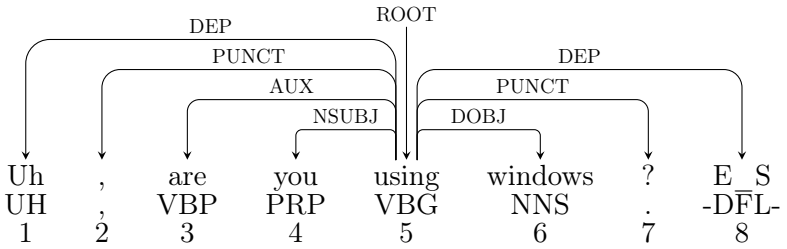

Figure 3.7: A tree that shows the typical usage of UH in the Switchboard corpus.

Another big cause of *dep* relations are the token tagged "UH", which marks hesitation, confirmation and similar markers that people utter in conversations to convey either awareness of receiving information from the speaker, thinking or confirmation.

The token "Uh" which is used as the first word in the sentence in Figure 3.7 marks that the speaker is trying to figure out something to say, and takes the turn by uttering "Uh". It also gives the speaker some time to think about what he wants to say, which would not be necessesary in a written context. A comma always follows the hesitations to note a pause, and it is most often connected to the root of the sentence.

E_S and N_S There is a E_S tag in the end of the sentence in the tree in Figure 3.7 that the reader might have noticed. We looked at the E_S and N_S tags in the Penn Switchboard corpus in Section 2.3. For the purpose of classifying utterances and making dependency trees, trying to reconstruct the complete sentence from the utterances would be difficult. Moreover, it is not very interesting for the dependency parsing, since broken utterances is something we would want to classify and data-driven dependency parsers are good at making trees that are not necesessarily a complete sentence.

Problematic roots

The initially converted Switchboard trees sometimes have the root token in some position which makes it necessary to move it. This happens in the cases where the root is a word tagged with the POS-tag "-DFL-" (most often the root will then be a \[) which is removed during the processing of the sentences.

It can also be that the root is in a position where it is deleted (placed between \[and \+). This happens most commonly where the entire utterance is in a reparandum. In this case, moving the root to an appropriate position might not be an option. We have not figured out a good way to deal with these trees, and there is a small number of them. These are part of the trees that are dropped shown in Table 3.4.

3.2.3 Speech Labels

The Stanford typed dependencies does not have an appropriate label for these relations (De Marneffe & Manning, 2008) since it was not written for spoken language. While the problem is noted by others and such annotation standards have been proposed for phrase structure grammars (Rehbein, Schalowski, & Wiese, 2012), the author has not been able to find any work that introduce any such annotation scheme for dependency grammars.

We therefore propose to augment the Stanford dependency function list with two additional dependency functions to describe the two key concepts that are noted in the surface form (Figure 3.3) and Part-of-Speech (Figure 3.7) in the Penn Switchboard corpus.

The two additional functions are:

- *Hesitation* The speech phenomena function marks word that are tagged as "UH" and consists of words like "Uh-huh", "uh" and "um".

- *Repair* The repair function replaces the \[, \+ and \] tags in the surface form, and encodes the repair and delete information into the dependency trees. It is encoded so that the first phrase is marked as a repair of the phrase behind the plus sign. In the case of a deletion, the repair is a relation to the left most word after the \]

Value	Count
Conversations	650
Sentences	110 504
Unique Sentences	72 272
number of "dep" functions	291 775
Bracketed Sentences	20 373
Uneven Bracketed Sentences	310

Table 3.2: Statistics on the Switchboard corpus after being processed by the Stanford converter.

Corpus overview

Table 3.2 provides an overview of the Switchboard corpus after it has been converted by the Stanford converter and before processing it further. The table also shows that about 1/3 of the corpus consists of utterances that have at least one identical utterance in term of surface form. These are mostly backchannel sentences, e.g. "Uh-huh . E_S" and "Yes . E_S".

We also want to remove as many of the "dep" functions as we can to get trees that are more informative in terms of syntactic functions. This is one of the motivations behind introducing the new labels and adding them to the dependency trees.

We propose processing the Stanford converter trees to add these labels using the algorithm shown in Algorithm 1. We will look closer at the algorithm in the following section.

3.3 Converting Disfluency Annotation

In order to capture the phenomena described in the previous section into the trees created by the Stanford converter, we propose an algorithm to post-process the trees.

We looked at a lot of different types of phenomena and trees in the last section. Here we will investigate in more detail how these different types of disfluencies might be processed to keep as much of the original structure from the Switchboard version as possible. This means that before removing the disfluency symbols, we have to make sure that we preserve the structure that is described in them.

The pseudo-code for this algorithm is shown in Algorithm 1. To keep the code simple, the algorithm does not include some of the corner cases. These corner cases will be mentioned in the text non the less. To ensure that we do not generate spurious trees we avoided the use of rules and heuristics that would create non-projective trees.

Algorithm 1 Pseudo algorithm for the conversion of the trees.

```
function connectRoot(t, root, dflPosition)
    nextWord ← searchForNextWord(t, dflPosition + 1)
    head(root) ← nextWord
    label(root) ← "repair"
end function
function moveAndLabelArcsHesitation(t, uhPosition)
    nextWord ← searchForNextWord(t, uhPosition + 1)
    head(uhPosition) ← nextWord
    label(uhPosition) ← "hesitation"
    if uhPosition + 1 = "," then
        head(uhPosition + 1) ← uhPosition
    end if
end function
function moveAndLabelArcsRepair(t, dflPosition)
    root ← findNewRoot(t, dflPosition)
    for word ← (dflPosition + 1) to |t| do
        if word = "+" then
            return root
        end if
        if head(word) = dflPosition then
            head(word) ← root
        end if
    end for
end function
function speechFunctionReplacement(t)
    rootStack ← empty
    if rootIsDFLSymbol(t) then
        t ← moveRoot(t)
    end if
    for word ← 1 to |t| do
        if word = "⌈" then
            push(rootStack, moveAndLabelArcsRepair(t, word))
            remove(t, word)
        end if
        if word = "+" then
            connectRoot(t, pop(rootStack), word)
            remove(t, word)
        end if
        if word = "⌉" or "N_S" or "E_S" then
            remove(t, word)
        end if
        if pos(word) = "UH" then
            moveAndLabelArcsSPF(t, word)
        end if
    end for
end function
```

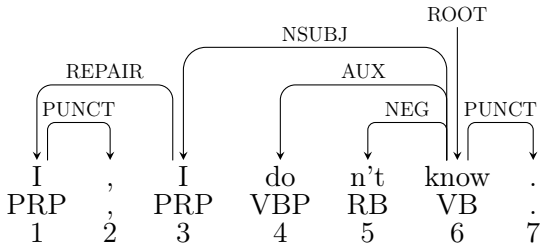

Figure 3.8: A finished tree with removed disfluency annotation.

3.3.1 Repairs, Duplications & Deletions

The removal of repairs, duplications & deletions are in many ways quite similar because they are annotated similarly in the original Switchboard Treebank. We will look at them as a common phenomena and then specify the exceptions that has to be performed for each of them.

Prepare The Reparandum

The first thing that needs to be done is reconnecting all the words that is attached to the restart annotation. This is so that we do not lose the structure that is in the annotation. For the reparandum, this means reattaching all the words inside the first part of the restart to a token outside the reparandum. This part is the same for all the phenomena annotated as restarts in the Switchboard corpus, namely repairs, duplications and deletions. In Algorithm 1 this part is done by the function named "MoveAndLabelArcsRepair".

For the basic duplication utterance, shown in figure 3.3, the reparandum is the tokens 2 and 3 ("I,"). Both tokens are connected to the "\[". If we remove the "\[" both the "I" and the "," would not be connected to the rest of the utterance. This would be problematic as it would either have to accept them as separate utterances or discard them, which we did not want to do originally.

The first step would be to reconnect token 2, "I", to the rest of the tree in such a manner that the original information on what is reparandum and what is the repair is preserved. We do this by locating the most probable root of the repaired part and connect it to the left most word in the repair.

The most probable root is found using the following heuristics. First look at which tokens are a dependent of the "-DFL-" token we want to remove. Select the first dependent that is not POS tagged as "UH" and does not have the function "PUNCT". If no such dependent exists select the first dependent POS tagged as "UH" to be the root. If there are dependents, select the first one.

The assumption made is that for the Switchboard trees that are correctly tagged there is only one token that is dependent on the disfluency symbol (i.e.

there is one phrase being repaired and not multiple ones), and if this is not case, the left most token is assumed to be the root.

Connect The Reparandum

Connecting the reparandum is also done in a similar manner for all the cases we are talking about. The only exception being the deletions which require some extra thought since they do not have a repair section of the restart to connect to.

Repair & Duplications The Repair section of the restart is often connected to the rest of the utterance with proper syntactic functions and with the correct head. This is natural, given that they are the part of the utterance that is supposed to be a part of the finished utterance. But this creates an issue in the ways we can reconnect the reparandum.

The issue is that we can not reconnect the words inside the repair to reflect what the repair really consists of. This is troublesome because then it does not become obvious what we are going to connect the reparandum to.

This problem is what the "connectRoot" function in the Algorithm 1 tries to solve. It uses the "searchForNextWord" function which simply searches for the next word-token inside the repair (i.e. after the "\+"), preferring non-dysfluency tokens.

Piecing it all together, the dependency tree in Figure 3.3 would be processed in the following manner:

- *Step 1, find a root in the reparandum:* there is only one word-token inside the reparandum and it is chosen as the new root by the "findNewRoot".
- *Step 2, connect all tokens in the reparandum to the root:* The comma which is connected to the "\[" is connected to the root in the reparandum (which is token 2, or "I" in this case).
- *Step 3, find a word-token outside the reparandum to connect to:* The "searchForNextWord" function finds the left most word-token in the repair (token 4, "I") and connects the reparandums root to that token using the "repair" function.
- *Step 4, remove all disfluency symbols:* Remove the "\[" "\+" and "\]" tokens.

The result is the dependency tree that we see in Figure 3.8, where "I" is the head of "I" with function label "repair".

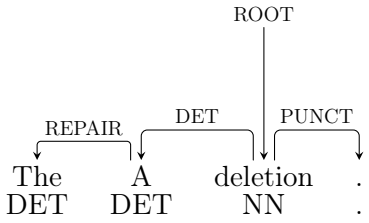

Figure 3.9: A deletion taken directly from the stanford converter output.

Deletions The Switchboard corpus marks a deletion as a restart that has a reparandum with no repair. A constructed example is shown in Figure 3.9. The removal of this structure is done in the same fashion as if it was a normal duplication or restart, only the "findNextWord" in the "connectRoot" function in Algorithm 1 looks outside the \ ] in order to find a suitable word-token to connect to. It is not marked as a deletion, but rather a repair because there is limited training data, and therefore it would be hard to train many new functions.

Uneven Brackets

We saw in Section 3.2 that utterances are sometimes split into more than one utterance and the brackets for the restarts end up being in different utterances in the treebank. While they require some special considerations, they are mostly handled like a normal restart.

The annotated trees that have these uneven restarts, in our approach fall into one of the following five categories.

I Contains a \ [and either a \ + in the last position or no \ + at all.

II Contains a \ [and a \ + where the \ + is not in the last position.

III Contains only a \ +

IV Contains a \ + and a \ ]

V Contains only a \ ]

In case I and V, we can not really connect the repaired or the repair phrase to anything, and therefore the disfluency symbol is checked to see if there are any other words connected to it. If there is anything connected to it, we connect everything to the root of the phrase. Unless it is a word tagged with "UH", in which case the rule on reconnecting hesitations, that we introduce later, takes presedence. Afterwards, the disfluency symbols are removed.

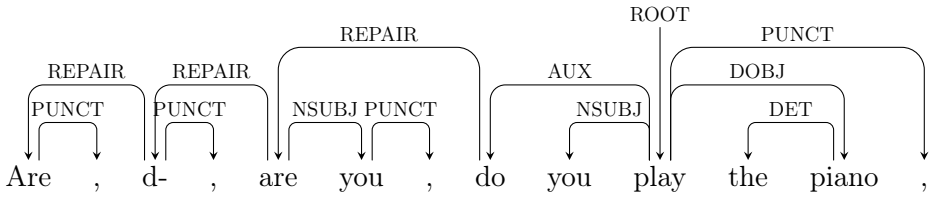

Figure 3.10: The dependency tree in Figure 3.6 after the post-processing with our algorithm.

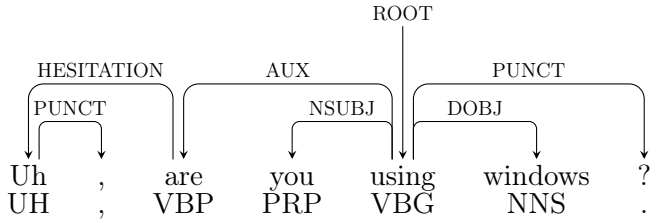

Figure 3.11: Removing the UH.

In the case of II, III, and IV the utterance is presumed to have the boundary symbols (either \[or \]) 1 position before or after the utterance, and then processed as a normal repair.

Nested Brackets

Sometimes the user repairs an utterance multiple times and the disfluency annotation is nested. The Algorithm 1 handles this by keeping a stack over all the roots and connects them from left to right, so the left most repair gets connected to the left side of the next repair in a recursive fashion.

In the finished dependency trees they become repairs of repairs etc. in the same number of subtrees as there where nested brackets. We can see this in Figure 3.10 where we have three edges labeled with "REPAIR", which is the same number of nested repairs that we see in the original tree in Figure 3.6.

Hesitations

The "UH" POS-tagged word-tokens are the tokens we want to represent with the "hesitation" function. The tokens that are marked with the "UH" POS-tag are words like "um", "uhu-uh" and similar.

The "UH" POS-tagged word-token is often followed by a punctuation of some sort; For the most part a comma. This is done to indicate that there is a pause after that word, but they are not directly related to sentence segmentation as we know from written text. Therefore it would be appropriate to say that the comma following the "UH" most likely has a direct relation to the "UH" token.

The comma or other "punct" related tokens are therefore connected to the "UH" token using the same function it had before it was reconnected. This also helps us avoid projectivity as the comma can be connected to the root, and there might have been a reconnect from a restart or similar earlier which would have made the structure projective.

The token itself is then connected to the word on the right of it regardless of which word it is to ensure non-projectivity in the tree and marked as a "hesitation" using the function by the same name. If it happens at the very end of the sentence, the "hesitation" is connected to the first token on the left instead.

Moving the root

Sometimes the sentences are short and there is no root in an utterance sense. In the Stanford trees, the roots are then assigned to the \[. We want to remove the disfluency symbols, so we have to move the root pointer to something other than a dysfluency symbol. This is done the same way as when we move the arcs. Find the first word that is preferably a non "UH" or "punct" word, and make it the root. Connect the rest of the dependents to \[to that word.

The E_S and N_S tags does not bring any information that is necessary in the context of the dependency parsing for spoken language, since a part of the task is to build trees even when they are not complete. They are for that reason simply removed from the output trees.

Unconverted Trees

Some sentences fall outside of the scope of the trees we can repair. The most common sentences that were not treated by the conversion algorithm are the ones where the root is in the deleted part of the repair (between \[and \+). The problem with these sentences is that the root is in need of moving, but there is no good place to move them to. They are also often accompanied by annotation errors and are generally hard to do anything with.

The other problem that comes up often is error in the dysfluency annotations, which is also hard to do anything with. And the last category is a rest category for things that does not go through the reference implementation.

These errors constitues about 2‰ of the corpus and are not a part of the produced treebank. The errors are further broken down in Figure 3.4.

3.3.2 Error Analysis

After the conversion we performed an error analysis in order to understand the behavior of our algorithm better. Table 3.3 shows that 279 sentences were left unconverted and Table 3.4 tells us the breakdown into categories about which errors these sentences belong to as described in the section Unconverted

What	Value
Conversations	650
Processed Sentences	110 255
Unconverted Sentences	249
Number of "dep" functions	42 687

Table 3.3: Statistics on the Switchboard corpus after being processed by the Post Processing algorithm for Switchboard trees.

Errors	# Affected
Errors with brackets	40
Root in Deleteted possition	169
Other types of errors	40

Table 3.4: Number of sentences that gets errors when run through the program, and what their errors are.

Trees. Table 3.4 shows that the most frequent error is that the root is found in the deleted root position.

There are still a lot of "dep" relations which should have been reduced, but the pattern of these relations are hard to find and there might be a need for many syntactically informed rules in order for the dep relations to be reduced even further. That is unfortunatly not in the scope of this thesis.

3.4 Training Dependency Parsers For Spoken Data

In this section we describe the setup that was used to investigate how well a parser trained on our post-processed corpus performed. This parser is compared to two other parsers trained on the Switchboard corpus processed in different ways and the Wallstreet Journal treebank.

In addition we look at how well the parser adapts to the spoken domain using parts of the Wallstreet Journal in combination with the different Switchboard corpora. We also investigate how much the spoken data deviates from the written data in the Wallstreet journal.

3.4.1 Parser Settings

The qualities that were key in choosing the parser settings were accuracy and speed, as a real-time spoken language parser would have to be fast as well as accurate.

The Maltparser is a package with several parsing algorithms. It was introduced in Section 2.2 and we see that it should be well suited for adapting to the spoken language domain because of its flexibility in adapting to new data and speed.

Algorithm	LAcc	UAS	LAS
Planar	91.75	85.96	84.26
Stack Eager	93.77	88.83	87.23
Stack Projective	93.79	88.84	87.25

Table 3.5: Different malt options tested.

The Maltparser package contains a lot of options, and the user for the large part has to take into consideration two aspects when choosing what options to use. These are which machine learning algorithm and parsing algorithm to use.

Machine Learning Algorithm

The first choice is the machine learning algorithm used for training the Maltparser. There are 2 major options to choose from, the liblinear vector machine implementation and the libSVM implementation.

The liblinear implementation is simpler, and for that reason much faster for big data sets. In spite of the simpler implementation, the accuracies between libSVM and liblinear for certain types of data are comparable. This holds for data with many features, like language data (Nivre & Hall, 2010; Cassel, 2009). Since libSVM is both quite a lot slower and has a low difference in accuracy for our particular field we chose to use liblinear and it is used during all our experiments in the current section.

Parsing Algorithm

Since we do not make a big point in this thesis about doing optimization, we chose the dependency algorithm by running a few parser algorithms on the Wallstreet Journal and then picked the one that performed best without any optimization. Table 3.5 shows the different algorithms tested. The table shows that the Stack Projective performed the best with a Labeled Accuracy Score of 87.25%. The Stack Projective algorithm was used in the experiments in this chapter.

3.4.2 Corpora

In this section we will describe the different corpora used in the experiments and what they consist of.

The Wallstreet Journal (WSJ) is a dependency treebank converted from the original WSJ revision 3 constituent treebank. It has been converted using the same converter and options as the Switchboard corpus, namely the Stanford parser with basic dependencies. The sections that are in use are sections 02 - 21, used for training. Section 23 is used for testing and Section 24 is the

```
((S
   (EDITED
     (RM (-DFL- \[))
     (NP-SBJ (PRP we))
     (, ,)
     (IP (-DFL- \+) ))
     (NP-SBJ-1 (PRP we))
     (RS (-DFL- \]))
   (VP (VBP try)
         (S (NP-SBJ (-NONE- *-1))
              (VP (TO to)
                   (VP (VB stay)
                        (PP (IN within)
                             (NP (DT a)
                                  (JJ certain)
                                  (NN budget)))))))
   (, ,) (-DFL- E_S)))
```

Figure 3.12: A tree taken out from the Penn Switchboard corpus showing the remove data from two of the SWBD corpora used. The gray area is removed in Charniak and the gray bold is kept in no-dfl

development test set named devel in the tables. This is the standard split used in most papers that use the Wallstreet Journal treebanks.

The first version of the Switchboard treebank was created using the conversion algorithm described in this chapter. The two other Switchboard corpora was created with swbd-transform, a script written by Jørgensen (2007) for removing the different types of disfluencies in the Constituent version of the Switchboard corpus. An outline of the different things that are removed using this script from a complete Switchboard constituent tree can be seen in Figure 3.12. The following versions are therefore used in our experiments:

- *SWBD Post-Processed:* The post-processed corpus is the corpus that we post-processed after the initial conversion with the Stanford converter from the original Penn Switchboard corpus. It is the result of Algorithm 1. See section 3.3 for more details.

- *SWBD Charniak:* The SWBD Charniak is created by using the "remove all disfluency" option in the swbd-transform script.[1] This corresponds to all the gray areas in Figure 3.12. It is then converted using the Stanford converter and used as-is. It is called Charniak because it would be this

[1]This is the -f option.

Corpus	Utterances	Tokens
WSJ Train	39832	950028
SWBD Post-Processed Devel	81629	773954
SWBD Post-Processed Devel Filtered	58959	734506
SWBD Charniak Devel	39869	349298
SWBD No-DFL Devel	58116	595676

Table 3.6: An overview of the size of the different treebanks used in the training.

kind of treebank that Charniak would have used in his approach which was described in the background section.

- *SWBD No-DFL:* The No-DFL version is the second corpus variant processed with the swbd-transform script. This version has all DFL tagged words and the "UH" Hesitations removed[2] before converting with the Stanford converter. In the example tree in Figure 3.12 this corresponds to removing all the gray text except the text that is in bold and up-right.

For training with the Switchboard 9/10 of the corpus was used. This corresponds to the conversations numbered 2504 up to 4936. The remaining conversations (2000 to 2503) were used as the test corpus.

Table 3.6 gives the reader an overview of the sizes of the different corpora in terms of the number of sentences and the number of word tokens in the different corpora. The reason why the same part of the Switchboard corpus in the different version differ so much is because of the removal process on the constituent trees illustrated in Figure 3.12 removes some word-tokens from the utterances' surface form.

The utterance counts differ because sometimes the entire utterance has been removed. For example the very normal feedback utterance "Uh-huh ." are removed in both the no-dfl and Charniak corpus because they are a disfluency token. Some utterances are contained within "(EDITED)", in which case it has been removed in Charniak but not entirely in no-dfl. Both these types of utterances is kept in the Post-Processed corpus.

Combining corpora

To get more data and a base that is the same in all the corpora, we used the Wallstreet Journal as a base and appended the Switchboard data on top of it. To make the treebanks roughly similar, the treebanks were normalized on word-tokens. This is to ensure that we can compare the spoken-language parsers when it comes to domain adaptation and how well they perform on the Wallstreet Journal corpus.

[2]This is the -s option.

Since we are interested in using these parsers for the downstream task of dialog act classification, we want to be able to compare parsers trained on written vs spoken language without the influence of data size.

The combining of WSJ and Switchboard was done using two tactics. One was to keep a predefined portion of the WSJ. These treebanks are noted with a k=<number> in the Table 3.7. The other was to use the entire Switchboard version of the treebank and splice it onto the Wallstreet Journal part, keeping only the amount of word-tokens from the Wallstreet Journal necessary to make the treebanks the same size.

The smallest treebank in terms of word-token count is the Switchboard Charniak bank, which can be seen in table 3.6. It consists of 349298 word-tokens. To make the WSJ base for the rest of the treebanks, the base became the size of the necessary data needed from WSJ to fill the rest of the 950028 word-tokens, or $950028 - 349298 = 600730$.

The rest of the combined Switchboard + Wallstreet Journal treebanks then used 600730 word-tokens from the Wallstreet Journal and 349298 from their own treebanks, rounding upwards towards the nearest whole tree in the treebank.

3.5 Results

The comparison of the parsers was done using four types of corpora. The Wallstreet Journal Devel and Test, Switchboard Charniak Test and the Switchboard Test corpora in the same annotation style as the training data [3]. The intuition behind testing on Wallstreet Journal is to see how far the spoken data has diverged from the written data found in Wallstreet Journal. The Switchboard tests are there to see how well the parsers adapted to the spoken language domain.

3.5.1 Testing on Wallstreet Journal

As stated in the previous section, to see how far the different parsers diverged from the data found in the Wallstreet Journal, we tested the different parsers on the Wallstreet Journal on the Test and Devel treebanks.

The training corpora were made using the merging tactic described in Section 3.4.2. The Wallstreet Journal Test and Devel treebanks are section 23 and 24 respectively as outlined in the same section.

Table 3.7 shows the results of the testing done on the Wallstreet Journal. Since we were more interested in keeping the settings for our parser the same for all of the training sets, no optimizing of the parsers were done. The baseline parser "WSJ" is for this reason not representing a state-of-the-art Wallstreet

[3] e.g. SWBD Post-Processed Test for the parser trained on SWBD Post-Processed

Malt Model	WSJ Test		WSJ Devel	
	UAS	LAS	UAS	LAS
WSJ	**88.84%**	**87.25%**	**87.64%**	**85.74%**
SWBD Charniak + WSJ	88.23%	86.59%	86.72%	84.78%
SWBD no-dfl + WSJ	87.24%	85.51%	86.05%	83.99%
SWBD no-dfl + WSJ k=600730	88.06%	86.40%	86.74%	84.77%
SWBD PP + WSJ	86.06%	84.28%	84.21%	82.01%
SWBD PP + WSJ k=600730	88.21%	86.56%	87.01%	85.08%

Table 3.7: The different parsers tested on the Wallstreet Journal Testing and Devel parts.

Journal. The selection was done by choosing the best of the 3 parser algorithms tested with default settings shown in Table 3.5.

The best scoring parser in Table 3.7 is the "WSJ" trained parser, with no additional information, having a labeled accuracy score of 87.25% and 85.74% for Test and Devel. This is not surprising as the "WSJ" parser is trained on more of the same type of written data that the test and devel sets consist of.

The best and the worst scoring, except the in-domain "WSJ" parser, are both post-processed trained parser. This is probably because as shown in Table 3.6, the SWBD Post-Processed treebank is the largest treebank of the three Switchboard corpora. Since all the treebanks are normalized to the same size using word-token counts, there is less Wallstreet Journal data in the one that does not preserve the Wallstreet Journal to a constant size.

Since the same drop is witnessed comparing the no-dfl treebanks, it seems reasonable to say that the spoken language data and the Wallstreet Journal data has diverging characteristics so that it makes an impact in how well the parser performs on written text.

Using the same number of training words for Wallstreet Journal and adding the Switchboard data shows that while the no-dfl approach performs the worst, there is little difference in how well they perform. With "SWBD no-dfl" at an average labeled accuracy score of 86.40% and 84.77% and the "SWBD PP" at 86.56% and 85.08%, one thing that can be said is that it is likely that the Switchboard data has some noteworthy differences compared to the Wallstreet Journal Data. While it might be interesting to know how well the treebanks conform with the Wallstreet Journal, this thesis is focusing more on parsing the speech data.

3.5.2 Testing on Switchboard

The main goal of a parser for spoken language is not to parse Wallstreet Journal data, and while the comparison with Wallstreet Journal data might be good for looking at how much the data deviates from written text, the primary goal is to have data available to parse spoken language. The experiments on the

Malt Model	SWBD Charniak Test		SWBD Test	
	UAS	LAS	UAS	LAS
WSJ	85.16%	80.13%		
SWBD Charniak + WSJ	**90.68%**	**88.32%**	**90.68%**	**88.32%**
SWBD no-dfl + WSJ k=600730	90.12%	87.61%	85.40%	82.06%
SWBD PP + WSJ k=600730	89.67%	87.07%	86.33%	84.14%

Table 3.8: The different training corpora on the charniaked Switchboard and their own respective training parts.

Switchboard corpora are divided into two sections. Testing on the speech data that has all disfluencies removed (the SWBD Charniak corpus) and the test corpus that has the same annotation style as the parser. Table 3.8 shows an overview of the results.

The parser is configured the same way that is described in the Section 3.4.1 and the treebanks are the same as in the previous sections.

Charniak

The second column in Table 3.8 shows all the training corpora tested on the charniaked switchboard corpus. The WSJ trained parser is also tested, since no extra information should be required for the parsing of a switchboard corpus that contains no disfluency information, and serves as a good reference point for how much speech data improved the overall performance of parsing speech data.

The "WSJ" trained parser performs worse overall on speech data than in its own domain with a difference in labeled accuracy of a little over 7 percentage points comparing Table 3.8 and Table 3.7. This shows that speech data, even when all the disfluencies are removed, do not correspond very well with what is seen in the traditional written corpus of Wallstreet Journal. Some domain adaptation is needed to improve the parser for parsing speech data.

The parser trained with the SWBD Charniak treebank shows a clear improvement over the parser trained on Wallstreet Journal only , with an improvement in labeled accuracy of 8 percentage points. This shows that training data from the spoken domain is indeed necessary. The reader might notice that it even performs slightly better than the Wallstreet Journal original tests, but that is not surprising given that the sentence length is quite a lot shorter in the Switchboard corpus than in the Wallstreet Journal corpus. Shorter sentences should be easier for the parser as the results seem to reflect.

The two other parsers with more speech data get a 0.7 percentage points and 1.3 percentage points drop off from the Charniak trained data. This is to be expected as the training data for these parsers differ from the data in the Charniak corpus. The parsers still parses spoken data better than the "WSJ"

Label	Recall	Precision
Repair	68.37%	82.68%
Hesitation	98.64%	98.44%

Table 3.9: The recall and precision for the new labels introduced in the Post-Processes corpus.

with a 6.9 percentage points improvement for the post-processed corpus that contains the new labels.

Parsing the same annotation format

The other part of Table 3.8 is the test on the same annotation format, and it is there to give a comparison on how the different parsers performs at parsing data in the same annotation style as their training data. The Charniak data is the same for this as in the previous paragraph. The interesting thing to compare here would be; how well did our approach do compared to the data that Charniak would have had in his approach and what happens if we only remove the disfluencies.

While the Charniak parser is, again, the best of the 3 with a good margin of 4.2 percentage points, it is again not that surprising, given that the Charniak data does not have anything more introduced than what is found in the Wallstreet Journal data, and in general have shorter sentences. Comparing the results of the post-processed data on the Charniak data with the run on its own test data shows this as well, since the labeled accuracy goes down by 2.9 percentage points.

The Post-Processed trained parser performs better than the no-dfl approach on its data. This indicates that the added labels are facilitating the parsing process. This in spite of the sentences being on average a little longer in the Post-Processed corpus than in the no-dfl corpus.

Parsing repairs and hesitations

As we recall, the post-processed parser introduced two new dependency labels to the Stanford label set in Section 3.2.2. Here we investigate how well the parser performs with these labels, and do a small error analysis.

Table 3.9 shows the precision and accuracy for the labels from the experiments shown in Table 3.8. While the hesitations got a high score of over 98%, it seems that the parser has some problems recognizing the repairs, in particular when it comes to recall.

That the hesitation is easy to pick up for the parser is not that surprising as it is a small sub-tree with little irregularities. All tokens tagged with the POS-tag "UH" is structured this way, which makes it easy for the parser to pick up where the structure should be.

The repair is a more complex structure where irregularities are more common because, as we recall, repairs includes repairs, duplications and deletions. The parser still manages to have quite a high precision at 82.68% for these structures, but it struggles with finding them and has a recall of 68.37%. The results are still quite encouraging given that the edit-detector system that Charniak and Johnson (2001) used in his experiments achieved about the same recall.

Chapter 4

Dialog Act Recognition

4.1 Motivation

As we discussed in the background chapter, classifying dialog acts is one of the goals for an NLU component in a dialog system. For humans, the task of dealing with the irregularities, intonation, gestures and other characteristics of spoken language is relatively easy; taking in all phenomena at once is not that easy for computers. In the context of dialog systems, the dialog act recognition helps the dialog manager determine the intent of the user.

This chapter describes a dialog act recognition system trained on the Switchboard corpus using machine learning. We will investigate the contribution of syntactic features in the task of dialog act classification. To investigate this we develop a system with features using no syntactic information. This system is then compared to a system augmented with features based on different types of syntactical information.

Section 4.2 provides an overview of the dialog act classifier, what information is provided and what information is generated. Section 4.3 describes the features used for the baseline classifier and discusses some of the findings using the baseline features. Section 4.4 describes the usage of a dependency parser trained on the corpus produced in Chapter 3 in our classification system.

4.2 System overview

In this section we are going to describe a system for doing dialog act classification using a machine learning algorithm. If we wanted to define the classification system as a "well-posed learning problem" as we did in Section 2.5 it would look like the following.

- Task T: Classifying a given utterance into a dialog act class.
- Performance measure P: The percentage of dialog acts assigned to the correct class.

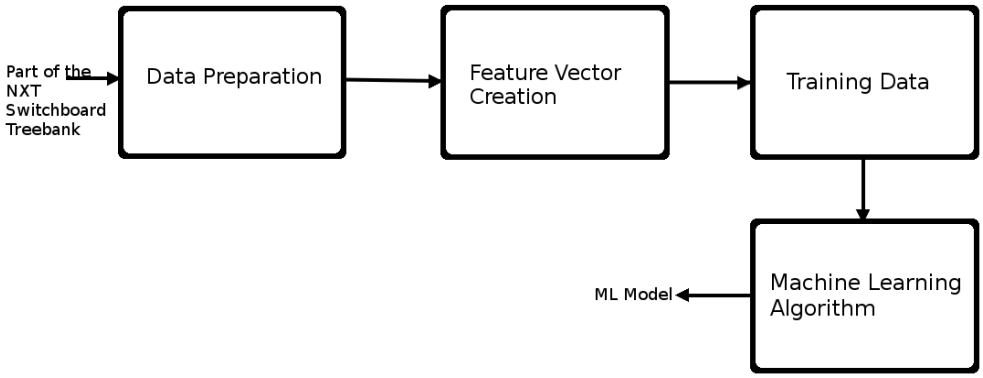

Figure 4.1: An overview of how the Vector Machine models was created.

Feature	Value
Turn Number	78
Speaker	A
Surface Form	You know or anything like that
POS Tags	PRP VBP CC NN IN DT
Dialogue Act	opinion

Table 4.1: An example of the features extracted from the NXT Switchboard corpus for one dialog act.

- Training experience E: Correctly tagged and classified features vectors.

We will present the system in three parts, the training of the model that the machine learning (ML) algorithm uses; the creation of the test data; and the setup of the dialog act tag prediction done by the machine learning algorithm. Figure 4.1 is an overview of the first part of the system, generating ML models.

4.2.1 Training of ML Models

In order to provide our models for dialog acts we use supervised learning. This means we train the machine learning algorithm with data annotated in the fashion we want and create a model using those data. Figure 4.1 shows a flowchart style overview of the development of the dialog act classifier from the NXT Switchboard corpus using Support Vector Machines (SVM). SVMs are described in Section 2.5. This section will focus on describing the input and output of this step of the system, training the ML model, and what data is provided.

The Input

The first step in generating our model is extracting the data that we want from the NXT Switchboard corpus. Here we will describe the data that we extracted from the NXT Switchboard corpus for use in the training of the ML models. The original data is represented in XML format and is a bit unpractical to read in its original form, but Table 4.1 shows an overview of the features we extracted for one sample dialog act. Here we will describe these features and what they represent.

The NXT Switchboard corpus is a corpus created by merging the data from the Penn Switchboard corpus and the MS-State transcript. The NXT Switchboard corpus is focused on being a resource for training dialog systems and include more information (e.g. prosodic information) than the Penn version. One of the more interesting parts for dialog act recognition is the dialog act annotation part of the corpus, where the NXT version has simplified dialog act tags that we described in Section 2.3.

This means that the NXT Swichboard corpus contains annotated dialog acts in the NXT format witch can be used for training Machine Learning systems to do dialog act recognition. The data extracted for each dialog act from the NXT Switchboard corpus includes the following:

Turn Number The first value in Table 4.1 is the turn number. The turn number represents the sequential order of dialog acts in the dialog this dialog act has. The turn number starts with 1 at the beginning of each conversation and goes up by 1 for each dialog act identified in the conversation. They are placed in the order of the time frame they occurred. Overlapping acts will be sorted by the one that started first.

Speaker Since the corpus contains transcriptions of phone conversations there is, for the most part, only two people in the dialog. The two speakers in the dialog is denoted by an A or B. If there occurs a dialog act from anyone but person A or B, as for example background noise, it is tagged with the nite tag third_pty.

Surface form The surface form is taken from the NXT Switchboard corpus, which corresponds to the Penn surface form.

The internal structure of the NXT Switchboard corpus allows for the mapping from the NXT dialog act tag to the surface form, but does not correspond to the syntactical structure of the sentence found in the Penn Switchboard corpus. This caused some problems later on, when trying to map from the trees produced in the chapter on Dependency Parsing of Spoken Language, which will be discussed later.

Part-Of-Speech tags Part-of-Speech tags taken from the NXT Switchboard corpus. The NXT corpus also takes the Part of Speech tags from its Penn counterpart, and they are therefore Penn style tags.

The data extracted from the NXT Switchboard corpus is then handed over to the Feature Vector Creation in a suitable format.

Dialog Act This is the NXT classification given to the current dialog act. The complete list of NXT classification labels and a description of what they mean can be found in the overview table in Table 2.1. The NXT tag is the target value for our classification system which we use as statistical measurement of how accurate our system is.

Feature Vector Creation & Training Data

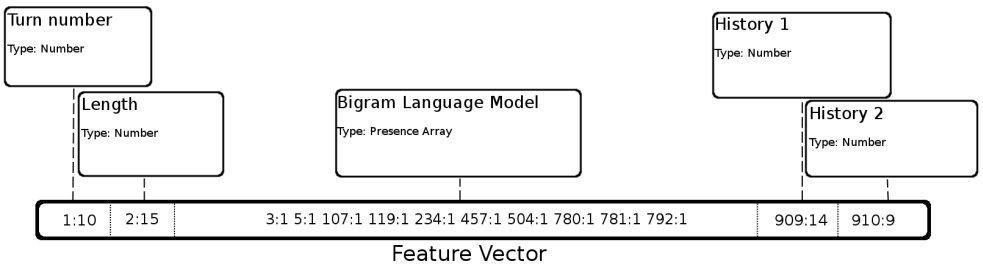

Figure 4.2: An example of a feature vector and how it may look like.

The Feature Vector Creation takes the data from the Data Preparation stage and creates feature vectors suitable for the Machine Learning algorithm targeted for the specific experiment. Figure 4.2 shows an overview of how a resulting vector may look like both on a abstract level and in its raw form. The colon separated numbers in the bottom of the figure represent the feature numbers and the values of those features. The feature vectors are often represented in a sparse way, so features that are not listed are presumed to have the value 0.

Features in this context are characteristics in the data belonging to a dialog act that we want to assign to the dialog act. E.g one simple feature may be, does the word "hi" appear in our utterance. If "hi" appears then the feature in our vector that contains this information is set to true. The goal of designing features is to create features that clearly distinguish each class of utterances so that a machine learning algorithm, like support vector machines, can say with confidence that this utterance is a part of a certain class of utterings.

The features we used to create the models will be described later in Section 4.3 and Section 4.4. Here we will describe the type of values that the features might take and the overall structure of a vector.

Binary Values The first and simplest value a feature can have is True or False. Our example feature "does "hi" appear?" is such a feature. These type of features will also be referred to as presence features, or features that check for the presence of a given condition.

Figure 4.2 shows a little more advanced version of this. The feature "Bigram Language Model" shows how a Language Model can be represented as an array of Binary values to capture slightly more complex relations. As the values are binary they will only take on the form of 0 or 1.

Natural Numbers When creating features that represents counts of phenomena or represent different classes, we use natural numbers. The value natural number features can take is an integer from 0 to N, where the N is implementation specific.

In Figure 4.2 there are four such features. The first in the vector with feature number 1, is the turn number feature. This is an integer that shows how many dialog acts has been uttered before this one, and it is represented as an integer.

We could create more complex features with this as well, e.g if we wanted to represent the counts of different bigrams in our "Bigram Language Model", we could use an array of natural number features instead of just binary values.

Some machine learning algorithms work better with binary values only or does not support number values. In this case we can convert the numbers into a binary form. This means using an array of binary values over multiple features instead of having one feature with a natural number. A lot of the features that we use that take natural numbers as value have undergone this conversion.

Training Data The resulting set of feature vectors are encoded in a suitable format and used as training data for the ML model.

Training The Model

The machine learning algorithm is trained on the resulting feature vectors in the training data. The result is a model of the features created that can be used by the machine learning algorithm to predict classes based on feature vectors. It is important that the feature vectors used in the prediction step has to be created in the same way as the training data so that the features in the vector represent the same things.

In our experiments we use Support Vector Machines (SVM) introduced in Section 2.5.2, more specifically libSVM and liblinear by Chang and Lin (2011). Since we are not interested in optimizing the SVMs but compare the different types of features, we use the default settings for both implementations in our experiments.

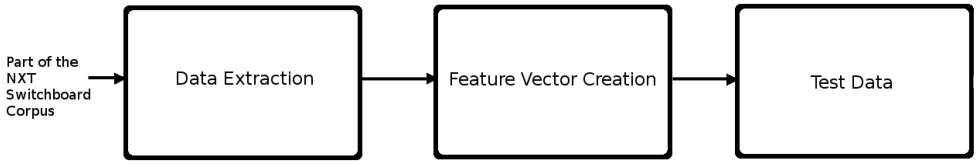

Figure 4.3: A flowchart showing how the test data is created.

	converstions
Devel	sw2504 – sw4936
Test	sw2005 – sw2503

Table 4.2: How the Switchboard corpus was split during the development and testing.

4.2.2 Creating Test Data

Figure 4.3 shows an overview of how our test data is generated. The reader might notice that the first two steps are exactly the same as in the previous section on creating the ML model. This is because to create the test data we take a different part of the NXT Switchboard corpus and run it through the same process as the training data. One should note that the features used in the generation of the test data are exactly the same features as those used in the creation of the training corpus. The result is a Data set with feature vectors that we can feed to the learning algorithm.

Data Split

For development purposes, the Switcboard conversations sw2504 to sw4936 was used, the rest (sw2005 to sw2503) were held out and used for the final testing described in Section 4.5.2. Table 4.2 shows an overview of this. The development was done using a 15-fold validation scheme. The validation was done on a conversation level, meaning that the 516 conversations remaning in the development part of the corpus was split into 15 equal pieces and the number of sentences might vary from chunk to chunk.

The setup produced 15 runs with the desired features and using either libsvm or liblinear as machinelearning algorithms. Different combinations of features where tested in both the linear and svm cases. In general, binary featues scored a higher accuracy with the linear classifier and multiclass features scored higher with libsvm. For the final results, only liblinear was used.

4.2.3 Applying The Model

The final step of the system is generating predictions over dialog act tags using the model we create in Section 4.2.1 on the feature vectors in the testing data

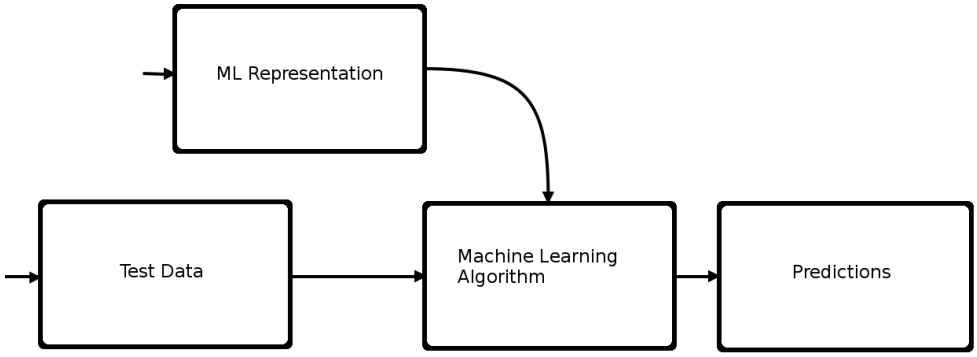

Figure 4.4: An overview of the final step in the system, creating the predictions.

described in Section 4.2.2. Figure 4.4 shows an overview of this process. The machine learning algorithm will find the best possible matches for our test feature vectors using the model.

4.2.4 Evaluation

The evaluation of our classification system is performed in four parts. First choosing a baseline classification system that contains no dependency information, then adding dependency features described in Section 4.4 using trees from three of the parsers trained in Section 3.5. The three parsers used are the "WSJ", "Charniak" and "Post-Processed" parsers.

Metrics

Here we will give an overview of the metrics used in describing the results of the systems. We will use TP, FP and FN for "True Positive", "False Positive" and "False Negative" respectively.

Precision Precision is the total number of correctly assigned tags for a dialog act divided by total number of dialog acts predicted for a given tag. It shows how sure we can be that a given tag is correctly labeled given that it is classified as a specific class.

$$Precision = \frac{TP}{TP + FP}$$

Recall Recall is the number of dialog acts actually being assigned the correct tag for a given class. It is calculated as follows:

$$Recall = \frac{TP}{TP + FN}$$

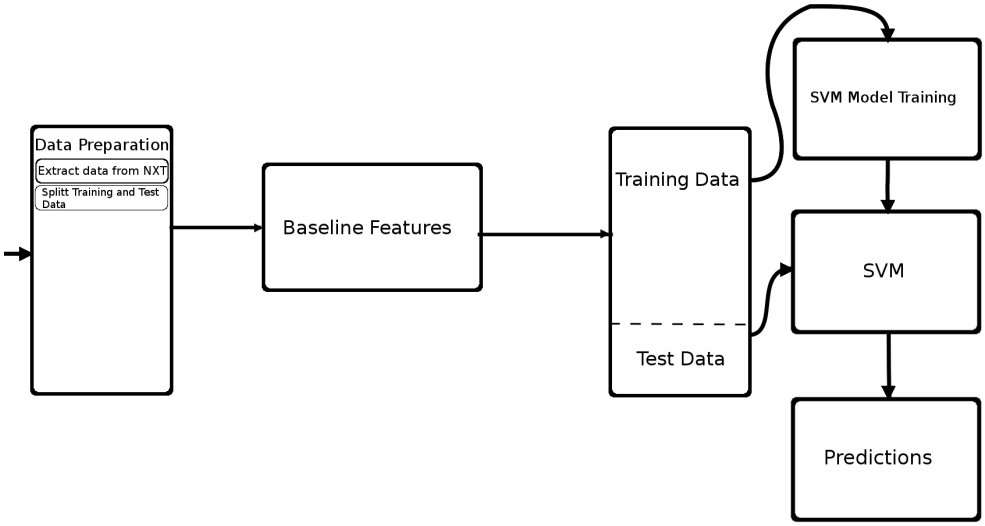

Figure 4.5: Our system as described in the previous section in one piece.

Accuracy The accuracy is used to measure the overall performance of the system. We do this by taking the total number of utterances classified into the correct class and divided by the number of classes.

$$Accuracy = \frac{TotalCorrectDAs}{TotalDAs}$$

F-Score the F_1-score is calculated using the following formula:

$$F_1 = 2 * \frac{Precision * Recall}{Precision + Recall}$$

We will refer to the F_1-score as simply the F-score for the remainder of the thesis.

4.3 Baseline

Figure 4.5 shows an overview of the entire system described in the previous sections in its entirety with a box for baseline features. In this section we describe the features that constitutes the baseline. We also look into some detail about how the baseline system performs before moving on to creating the syntactically informed features.

4.3.1 Baseline Features

Features are, as mentioned in the previous section, characteristics given to objects. In our case the objects are dialog acts. Here we will describe the

features that we designed to do classification of spoken utterances from the NXT corpus using no form of syntactic information.

- *Turn Number:* The number of dialog acts in the current conversation preceding this one.
- *Length:* The length or token count of the utterance.
- *History:* The previous dialog act.
- *History Trigrams:* The two previous dialog acts.
- *Unigram models:* Presence of words.
- *Bigram models:* Bigram language model with presence of bigrams.
- *Filtered unigram models:* Presence of the N-most frequent words.
- *Filtered bigram models:* Bigram language model filtered to the N-most frequent.

Turn Number

Turn numbers are constructed on the sequence of dialogue acts that are counted in a conversation. A turn number then roughly corresponds to the turns taken in the structure described in Section 2.4.1, but also includes backchannel responses that might happen during the other speakers turn. They are sequentially numbered from 1 to the length of the conversation, and are the same as the ones found in the input data described in the previous section. The feature is represented with a natural number.

Length

The length is the sum of the amount of surface level tokens found in a given utterance. The tokens are tokenized the same way as the NXT Switchboard corpus. The feature is also represented as a natural number.

History

N	DA	History	Sentence
#1	open	nil	Hi.
#2	open	open	Hi.
#3	yn_q	open	Have you seen my cat?
#4	yes	yn_q	Yes.

Table 4.3: Example conversation with history feature.

History is the class number of the previously act. Table 4.3 shows how this might look like. The feature is not restricted in how many of the previous act where used, but in this thesis a maximum of two where used. Due to time constraints, the history also use the gold standards found in the input. It produces one or two natural number features depending on how many of the previous dialog acts it keeps track of.

History trigrams

The history trigram is a natural number feature that has one number for each pair of previous dialog acts. For our example conversation in Table 4.3, utterance #4 would have gotten the id for the bigram (open, yn_q) to predict the next dialog act. It uses start and end tags to keep track of the start and the end.

Unigram models

The unigram models are presence arrays of word-tokens found in the training data. Tokens that are not found in the training data are given a default value for unknown token. The tokens are the same as the ones found in the NXT Switchboard corpus. For the sample conversation in Table 4.3, the first utterance would have had the feature for "Hi" set to True.

Bigram models

Bigram models are created over the surface form of the dialogue act phrases found in the NXT corpus. Start and end tags were added and the symbolic value for these were two consecutive dollar signs ($$). The tokens here are also the same tokens as the ones found in the NXT Switchboard corpus.

The finished bigrams would look like "($$, the), (the, dog), (dog, barked), (barked, $$)" if we took the phrase"The dog barked" and made bigrams out of it. The feature produces an array of natural numbers where the number represent the bigram found and the length of the array is the number of bigrams in the sentence. If the bigram is unknown, the corresponding feature has a value of 0.

Filtered unigram

The filtered versions of the unigram model only takes the N most frequent tokens in the training data and assigns one feature per of those tokens. That means if we set the filter to 50, we get a presence array with the length 50 where the 50 features represents the 50 most frequent tokens in the training data.

Filtered Bigrams

The filtered bigrams work in much the same way as the bigram model feature with regards to tokenising them and creating the bigrams. Like the filtered unigrams, this feature will also keep only the N most frequent bigrams found in the training data.

Filtered Bigrams differ from the Bigram models in that it creates a Presence array for the most frequent bigrams and will only check for the presence of the N most frequent bigrams instead of creating one feature for each bigram position in the utterance.

4.3.2 Baseline Results

Here we examine the feature selection for the baseline. We also describe how we arrived at using those features in a short manner and the settings used for the baseline features.

Selecting features

The features were added one at a time and checked to see if they improved the system. An increase in accuracy meant the feature was a candidate for our baseline system. For the features that were filtered, the N was decided running the features with different values of N together with the candidate baseline features.

In the end, not all features where chosen. The filtered versions of the Unigrams and Bigrams worked better than keeping all of them. The features that scored the best and chosen for our baseline was

- *Turn Number*
- *Length*
- *History* with two history elements
- *History Trigrams*
- *Fitered Unigrams* with the 500 most frequent words.
- *Filtered Bigrams* with the 1500 most frequent bigrams.

Results

Table 4.4 shows the result of the baseline system. In total, the baseline system has a 74.63% accuracy. The parser seems to handle tags that has small syntactic structures, like straight "yes", "no" and "backchannel" with an F-Score of 0.85+ for each of them. Most of the other tags had F-Scores lower than 0.70, even for large classes like opinion. This seems to be because the system puts

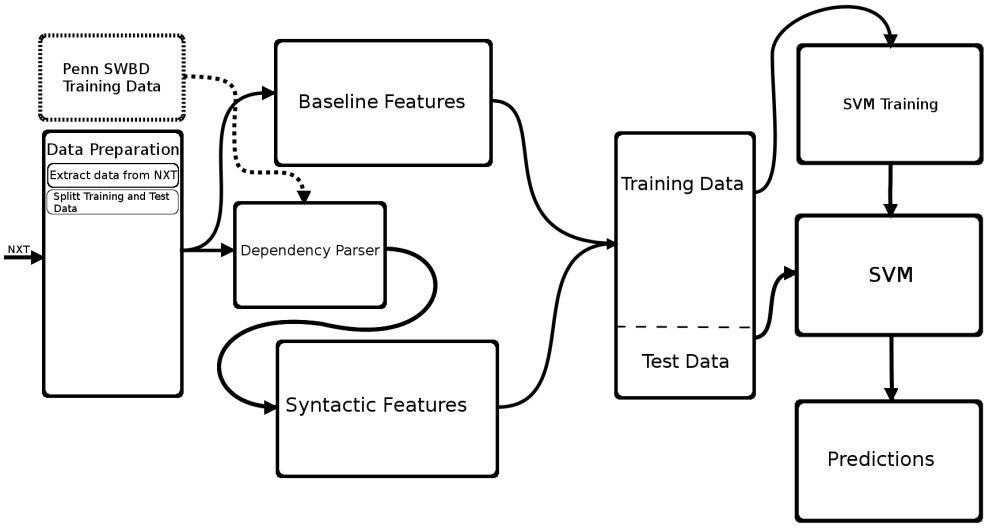

Figure 4.6: An overview of the system setup with dependency features.

many of the longer utterances in the "statement" category because it is the largest category and the machine learning algorithm has optimized on putting them there.

4.4 Dependency Based Features

The main question of this thesis is "does the dependency features add anything statistically significant to the classification task?". We investigate this by comparing the baseline system described in Section 4.3 with no dependency features with the system described here that uses syntactic dependency trees. We also look into a little more detail what has happened to the different classes in Section 4.5.1 with this comparison.

For this purpose we have modified our system to include trees from a depdency parser. In this case it is the Maltparser that we introduced in Section 2.2. Figure 4.6 shows the modifications of the baseline system shown in Figure 4.5. The figures shows an added box for syntactic features which takes dependency representations from a Dependency Parser. In the following section we will describe the features proposed for the syntactically informed features and how they where selected for use.

We also look at the prospect of different types of disfluency annotation affecting the results. The comparison is done using the same system and feature selection only changing the dependency parsers. The dependency representations are produced by the three parsers mentioned in Section 3.5, "WSJ", "Charniak" and "Post-Processed". This should give an insight into whether the domain adaptation helped comparing "WSJ" and "Post-Processed". It should

NXT tag	Gold	Predicted	TP	TN	FP	Recall	Precision	F-Score
yn_q	2215	1996	1427	788	569	64.42%	71.49%	0.68
yn_decl_q	565	219	58	507	161	10.27%	26.48%	0.15
yes	1287	1316	1132	155	184	87.96%	86.02%	0.87
wh_q	934	955	668	266	287	71.52%	69.95%	0.71
uninterp	1209	603	272	937	331	22.50%	45.11%	0.30
third_pty	46	22	6	40	16	13.04%	27.27%	0.18
thank	26	6	1	25	5	3.85%	16.67%	0.06
tag_q	18	2	0	18	2	0.00%	0.00%	0.00
sum	451	160	44	407	116	9.76%	27.50%	0.14
statement	34620	38862	30561	4059	8301	88.28%	78.64%	0.83
self_talk	31	15	2	29	13	6.45%	13.33%	0.09
rhet_q	240	130	33	207	97	13.75%	25.38%	0.18
repeat_q	108	38	12	96	26	11.11%	31.58%	0.16
repeat	294	37	4	290	33	1.36%	10.81%	0.02
reject	160	62	37	123	25	23.13%	59.68%	0.33
quote	411	199	96	315	103	23.36%	48.24%	0.31
other	323	251	191	132	60	59.13%	76.10%	0.67
or	90	73	49	41	24	54.44%	67.12%	0.60
opinion	12488	11730	7837	4651	3893	62.76%	66.81%	0.65
open_q	314	287	194	120	93	61.78%	67.60%	0.65
open	94	105	69	25	36	73.40%	65.71%	0.69
no	554	567	495	59	72	89.35%	87.30%	0.88
neg	132	108	46	86	62	34.85%	42.59%	0.38
maybe	42	9	1	41	8	2.38%	11.11%	0.04
hold	258	175	113	145	62	43.80%	64.57%	0.52
hedge	540	502	372	168	130	68.89%	74.10%	0.71
excluded	489	222	137	352	85	28.02%	61.71%	0.39
downplay	30	24	11	19	13	36.67%	45.83%	0.41
directive	310	199	86	224	113	27.74%	43.22%	0.34
decl_q	40	14	3	37	11	7.50%	21.43%	0.11
completion	250	55	11	239	44	4.40%	20.00%	0.07
commit	48	21	5	43	16	10.42%	23.81%	0.14
close	817	816	738	79	78	90.33%	90.44%	0.90
backchannel_q	515	566	358	157	208	69.51%	63.25%	0.66
backchannel	14412	16332	13272	1140	3060	92.09%	81.26%	0.86
apprec	1926	1627	1225	701	402	63.60%	75.29%	0.69
apology	29	6	2	27	4	6.90%	33.33%	0.11
answer	138	90	29	109	61	21.01%	32.22%	0.25
ans_dispref	109	42	5	104	37	4.59%	11.90%	0.07
agree	4725	3276	2182	2543	1094	46.18%	66.61%	0.55
affirm	376	349	158	218	191	42.02%	45.27%	0.44
acknowledge	636	474	300	336	174	47.17%	63.29%	0.54
abandon	5492	5250	3284	2208	1966	59.80%	62.55%	0.61

Table 4.4: Complete table for the run with all the baseline features.

1	I	_	PRP	PRP	_	2	repair	_	_
2	but	_	CC	CC	_	4	cc	_	_
3	I	_	PRP	PRP	_	4	nsubj	_	_
4	was	_	VBD	VBD	_	0	root	_	_
5	from	_	IN	IN	_	4	prep	_	_
6	conservative	_	JJ	JJ	_	7	amod	_	_
7	Wisconsin	_	NNP	NNP	_	5	pobj	_	_

Figure 4.7: A dependency tree taken out of our training data.

also show how much effect the disfluency annotation had by comparing "Charniak" with "Post-Processed".

4.4.1 Creating Dependency Trees

Using the trees from the Penn Switchboard corpus directly for the dialogue acts found in the NXT Switchboard corpus was not an option. This is because the utterances extracted from the NXT Switchboard using the dialog acts and the syntax trees in the Penn Switchboard corpus are on different levels of analysis and do not directly correspond to each other. This results in a difference between what one utterance in Penn Switchboard is and what one dialogue act is in the NXT corpus. While it is correct that a complete syntactical unit might be more than one dialogue act it also means we can not extract the trees directly from our Switchboard corpus that are described in Chapter 3. This is because neither does the sentence number correspond to the dialogue act order in the NXT corpus nor does the surface forms correspond to each other when we extract the data from the NXT by using the Dialog Act tags. A reconstruction of the trees from the Switchboard corpus to match the dialogue acts in the NXT corpus would require a deeper syntactical and semantic inspection than what is the scope of this thesis.

In order to create the dependency representations used in the classification, a Maltparser model trained on the Penn Corpus, the Wallstreet Journal training part and the processed Switchboard development part, were used. Each dialogue act that was found in the NXT corpus were run through the parser, and the resulting trees were input to the feature extraction for the machine learning algorithms.

4.4.2 Syntactic Features

Here we describe the syntactic features used in the second round of dialog act classification. We will try to relate all the features to the values it would have taken from the dependency graph in Figure 4.7. The dependency features employed in our approach are extracted from the following:

Dependency root

The root of a sentence is often a good indication of what we are trying to convey. The first feature examines the root of the dependency tree, which is the root of the utterance, numbers the surface form of the root word and returns the appropriate feature vector. If the specific root is not found in the training, the value of the root becomes 0. For the tree in Figure 4.7 this would have been the word-token 'was'.

The dependency root feature takes the surface form of the word that is pointed out to be the root of the dependency tree. Each surface form is given its own number, e.g. 'walk' and 'walking' would have different numbers because they do not have the same surface form.

Presence of labels

The presence of labels is a presence array representing the presence of a given type of dependency label. The list of lables are the 54 labels that are found in the Stanford dependency manual (De Marneffe & Manning, 2008), plus the two additional labels that were introduced in section 3. For the tree in Figure 4.7 this would mean that the features representing "repair", "cc" "nsubj", "root", "prep", "amod" and "pobj" would have been assigned the value 1.

Direction of edges with specific labels

The direction of edges feature is a feature that expresses whether the node's head is left or right of the current node for specific labels. This is achieved by looking at the number of the head and checking if it is lower or higher than the current nodes number. The resulting feature-vector is a presence array with true and false values for if a label goes left and one for if the label goes right. The number of features would for that reason be twice as many as the previous "Presence of labels" feature.

Number of occurrences of different labels

This feature creates an array of natural numbers with the counts of the different labels within the utterance for the given dialog act. It is similar to the Presence of labels only giving counts instead of simple presents values. In our example dependency tree in Figure 4.7, this feature would have the same values as the "Presence of labels" feature. But if the speaker had e.g. stuttered another I before the sentence and created another arc labeled "repair", the feature representing "repair" would have had a 2 instead of 1.

Label having POS as value

The Label having POS as value feature assigns one feature for each combination of label and POS-tag, giving it a theoretical maximum of $54 * 60$ number

of features. It collects and numbers all the found relations, and assigns each number a place in the feature vector, giving the respective place a 1 or a 0 for the presence of that label to POS pair. For Figure 4.7, the relations would have been {(PRP, repair), (CC, cc), (PRP, nsubj), (VBD, root), (IN, prep), (JJ, amod), NNP, pobj)}, and the features representing these would have been set to 1.

Label having surface form as value

This feature is similar to the "Label having POS as value" feature, but using surface form as the target of the label instead of the POS value. This means that the pair (PRP, repair) in the previous feature would have been ("I", repair) in this.

This feature can become quite large and for that reason also has a filtering option. The filtering uses a cut-off solution where only combination seen over a certain threshold of times are included in the feature vector.

4.4.3 Selecting Dependency Features

Selecting the dependency features happened in the same manner as the baseline features. They were added to the baseline one at a time and the results examined to see if they improved the system. If they improved the system they where added as a candidate for the dependency system. For the "label having surface form as value" feature that had a "cut-off" option, the "cut-off" was increased until the improvement in accuracy reached a top. There was only two dependency features that were used in the final system. These were:

- *label having POS as value*
- *label having surface as value* with a cut-off of 750

4.5 Results

This section is broken down into three sections. First we examine at the overall results and how well the different classifiers with the different features performed. Then we look a little more in detail on the systems results, breaking the results down by class. In the last section we look at the experiments with the held out data.

4.5.1 Overall Results

The overall results were obtained by running a 15-fold run on the 9/10 of the Swichboard corpus that we used and described in Section 4.2 section. The accuracies are measured by averaging over all the 42 classes used in the nite classification and averaging over the 15 runs done. The four system tested are

Classification Setup	Accuracy
Baseline	74.630%
Post-Processed Dependency Trees	**75.255 %**
WSJ Dependency Trees	75.101 %
Charniak Dependency Trees	75.250 %

Table 4.5: The total accuracy after a 15-fold validation.

- *The Baseline* with no dependency information used.
- *Post-Processed (PP)* A parser that is trained on the converted corpus proposed in Chapter 3.
- *Wallstreet Journal (WSJ)* A parser where the dependency parser is trained on the Wallstreet Journal only.
- *Charniak* A parser that is trained on the converted corpus with removed disfluency in the data, the same used in Chapter 3

The baseline uses most of the features described in the section 4.3, except the unfiltered bi-grams and uni-grams. The filtered variant of the N-grams used the 500 most frequent uni-grams and the 1500 most frequent bi-grams. The baseline is a respectable 74.63%.

The Post-Processed in Table 4.5 is the result of an experiment with the corpus that we converted and post-processed to contain disfluency tags in Dependency Parsing of Spoken Language. It performs 0.825 percentage points better than the baseline.

The last result (Charniak) in Table 4.5 is made with using parse trees from a Maltparser without any disfluency annotation in it. This means that the parser does not produce the "repair" and "hesitation" function labels introduced in Chapter 3. The features used in this experiment are the same ones as those found in the last one with disfluencies. This is done to test the hypothesis that the extra disfluency tags are aiding in some way in the classification. Table 4.5 show that the difference is 0.005 percentage points, which is a very small difference. This might be attributed to the fact that we have no features that directly take use of these tags.

Statistical Significance

We wish to assess the statistical significance of the differences observed in the previous section. An overview of the P-values for four systems compared with a paired T-Test over the 15-fold runs can be seen in Table 4.6.

The paired T-Test shows that the difference between the baseline and the dependency based system is statistically significant with a significance score

Opposing classifiers	P-Value
Baseline & Post-Processed Dependency Trees	0.000
Charniak & Post-Processed Dependency Trees	0.760
WSJ & Post-Processed Dependency Trees	0.104

Table 4.6: Paired T-Test relevancy score with 15-folds.

(α) of 0.01. While there is only about 0.6 percentage points difference on average between the disfluency and baseline system, the differences were consistent in all the 15 folds.

While the SWBD PP systems, compared to the baseline, made a statistically significant improvement, the addition of the dependency relations "repair" and "hesitation" does not seem to have added any significant to the parses, and the two Switchboard systems compare quite similarly.

The WSJ experiment shows that a WSJ only parser will improve overall quality of the classification. Given that the margins from the baseline is so small (0.6 percentage points) the small improvement achieved with WSJ parses is enough to make the added spoken language data statistically insignificant for the task of classification. Even if the difference between the "WSJ" parser and the "SWBD" parsers are not statistically significant, the domain adapted parsers still perform marginally better.

Error analysis

Here we investigate, in more detail, what the improvements consists of. Comparing the baseline and the Post-Processed approach, we look for any improvement in one system over the other to see what the improvement from the baseline to the dependency features consists of. Table 4.7 shows a compressed table with Precision, Recall and F-Score for the classes in our classification for the baseline and Post-Processed runs.

Going through the table, most of the large classes like "statement", "backchannel" and "opinion", the F-score only varies with about 0.01. But what might be more interesting is that in the same large classes, the precision of the system using dependency features is higher while the recall is lower, e.g for the largest class "statement", the recall has dropped from 88.28% to 87.63% but the precision has gone up from 78.64% to 80.05%.

The classes that seem to counter this positive increase in accuracy is the classes that are context sensitive and needs information that goes beyond the sentence-level, e.g The class "sum" (short for "Summary", see description in Table 2.1), needs to know that the information repeated in the utterance has been given before in order to recognize that this is a summary. Hence, these are difficult to classify correctly. This is shown in that the classes "sum" goes down from 27.5% to 20.42%. Another similar class "completion" goes down from 20% to 13.58%. The class "repeat" is also interesting as it is one of the classes that

Nite tags	Recall	Precision	F-Score	Recall	Precision	F-Score
yn_q	64.42%	71.49%	0.68	63.93%	71.34%	0.67
yn_decl_q	10.27%	26.48%	0.15	12.04%	25.66%	0.16
yes	87.96%	86.02%	0.87	87.65%	86.04%	0.87
wh_q	71.52%	69.95%	0.71	71.41%	72.03%	0.72
uninterp	22.50%	45.11%	0.30	25.56%	45.64%	0.33
third_pty	13.04%	27.27%	0.18	15.22%	36.84%	0.22
thank	3.85%	16.67%	0.06	15.38%	36.36%	0.22
tag_q	0.00%	0.00%	0.00	0.00%	0.00%	0.00
sum	9.76%	27.50%	0.14	8.65%	20.42%	0.12
statement	88.28%	78.64%	0.83	87.63%	80.05%	0.84
self_talk	6.45%	13.33%	0.09	9.68%	16.67%	0.12
rhet_q	13.75%	25.38%	0.18	15.00%	26.87%	0.19
repeat_q	11.11%	31.58%	0.16	14.81%	32.65%	0.20
repeat	1.36%	10.81%	0.02	4.42%	23.64%	0.07
reject	23.13%	59.68%	0.33	22.50%	49.32%	0.31
quote	23.36%	48.24%	0.31	25.55%	48.17%	0.33
other	59.13%	76.10%	0.67	59.44%	73.28%	0.66
or	54.44%	67.12%	0.60	56.67%	66.23%	0.61
opinion	62.76%	66.81%	0.65	62.70%	67.64%	0.65
open_q	61.78%	67.60%	0.65	60.83%	69.20%	0.65
open	73.40%	65.71%	0.69	78.72%	71.15%	0.75
no	89.35%	87.30%	0.88	89.71%	87.35%	0.89
neg	34.85%	42.59%	0.38	31.06%	42.71%	0.36
maybe	2.38%	11.11%	0.04	2.38%	9.09%	0.04
hold	43.80%	64.57%	0.52	42.25%	65.27%	0.51
hedge	68.89%	74.10%	0.71	67.96%	71.26%	0.70
excluded	28.02%	61.71%	0.39	27.40%	60.36%	0.38
downplay	36.67%	45.83%	0.41	36.67%	50.00%	0.42
directive	27.74%	43.22%	0.34	29.68%	43.60%	0.35
decl_q	7.50%	21.43%	0.11	7.50%	23.08%	0.11
completion	4.40%	20.00%	0.07	4.40%	13.58%	0.07
commit	10.42%	23.81%	0.14	10.42%	19.23%	0.14
close	90.33%	90.44%	0.90	89.72%	91.63%	0.91
backchannel_q	69.51%	63.25%	0.66	70.68%	62.22%	0.66
backchannel	92.09%	81.26%	0.86	92.22%	81.66%	0.87
apprec	63.60%	75.29%	0.69	68.12%	72.25%	0.70
apology	6.90%	33.33%	0.11	17.24%	71.43%	0.28
answer	21.01%	32.22%	0.25	22.46%	37.80%	0.28
ans_dispref	4.59%	11.90%	0.07	5.50%	14.29%	0.08
agree	46.18%	66.61%	0.55	48.04%	66.71%	0.56
affirm	42.02%	45.27%	0.44	40.69%	44.22%	0.42
acknowledge	47.17%	63.29%	0.54	47.17%	63.69%	0.54
abandon	59.80%	62.55%	0.61	69.34%	64.55%	0.67

Table 4.7: Table with all the classes comparing the baseline to the post-processed corpus.

needs context. While it only has a recall of 4.42% in the dependency system, it actually increases from an F-score of 0.02 to 0.07.

Some of the small classes have got a drastically higher score than its baseline. The class "thank" e.g. only appears 26 times in the corpus, but has gotten improved from an F-score of 0.06 in the baseline classifier to a score of 0.22 with dependency features. The class "apology" has also improved its baseline with more than double, but is still a small class of only 29 occurrences. Since they are small classes, this might mean either that the machine learning algorithm needs less training to spot these classes with syntactic info, or it might be a coincidence given the small number of the classes.

Table 4.8 shows a rundown of all the classes with the best run and dependency features. What is interesting to see in this table is that many of the classes that score the worst in the classification are classes that need the contextual information. These include the classes we looked at earlier ("sum", "completion" and "repeat"), but also other classes that needs other types of contextual information like "third_pty" that needs to know there is another speaker that is uttering something.

In this section we observe some improvement with using dependency features, especially pertaining to accuracy of some classes. The drops that was most noticeable in our large comparison tables were mostly classes that need more context information, like "sum", in order to be classified correctly.

4.5.2 Testing on Held-Out Data

Classification Setup	Accuracy
Baseline	75.12%
WSJ Dependency Trees	**75.65%**
Charniak Dependency Trees	75.59%
Post-Processed Dependency Trees	75.48%

Table 4.9: The results from the classification on the held out data.

In the previous section, the "WSJ" system had a disadvantage in that the system for Switchboard used a dependency parser partly trained on the same data that was the input of the system. In this section we examine the three systems tested on the held-out data we talked about in Section 4.2.2 (sw2005 – sw2503).

Table 4.9 shows the result of the experiments on the held-out data. In this test the system that performed the best is the system with data from the "WSJ" parser with an accuracy of 75.65%. Given that the gap between the systems is small, it would be hard to provide any hard conclusions, but it does not look like the unfair advantage of the Switchboard systems had that much

Nite tag	Gold	Predicted	TP	FN	FP	Recall	Precision	F-Score
yn_q	2215	1985	1416	799	569	63.93%	71.34%	0.67
yn_decl_q	565	265	68	497	197	12.04%	25.66%	0.16
yes	1287	1311	1128	159	183	87.65%	86.04%	0.87
wh_q	934	926	667	267	259	71.41%	72.03%	0.72
uninterp	1209	677	309	900	368	25.56%	45.64%	0.33
third_pty	46	19	7	39	12	15.22%	36.84%	0.22
thank	26	11	4	22	7	15.38%	36.36%	0.22
tag_q	18	2	0	18	2	0.00%	0.00%	0.00
sum	451	191	39	412	152	8.65%	20.42%	0.12
statement	34620	37902	30339	4281	7563	87.63%	80.05%	0.84
self_talk	31	18	3	28	15	9.68%	16.67%	0.12
rhet_q	240	134	36	204	98	15.00%	26.87%	0.19
repeat_q	108	49	16	92	33	14.81%	32.65%	0.20
repeat	294	55	13	281	42	4.42%	23.64%	0.07
reject	160	73	36	124	37	22.50%	49.32%	0.31
quote	411	218	105	306	113	25.55%	48.17%	0.33
other	323	262	192	131	70	59.44%	73.28%	0.66
or	90	77	51	39	26	56.67%	66.23%	0.61
opinion	12488	11576	7830	4658	3746	62.70%	67.64%	0.65
open_q	314	276	191	123	85	60.83%	69.20%	0.65
open	94	104	74	20	30	78.72%	71.15%	0.75
no	554	569	497	57	72	89.71%	87.35%	0.89
neg	132	96	41	91	55	31.06%	42.71%	0.36
maybe	42	11	1	41	10	2.38%	9.09%	0.04
hold	258	167	109	149	58	42.25%	65.27%	0.51
hedge	540	515	367	173	148	67.96%	71.26%	0.70
excluded	489	222	134	355	88	27.40%	60.36%	0.38
downplay	30	22	11	19	11	36.67%	50.00%	0.42
directive	310	211	92	218	119	29.68%	43.60%	0.35
decl_q	40	13	3	37	10	7.50%	23.08%	0.11
completion	250	81	11	239	70	4.40%	13.58%	0.07
commit	48	26	5	43	21	10.42%	19.23%	0.14
close	817	800	733	84	67	89.72%	91.63%	0.91
backchannel_q	515	585	364	151	221	70.68%	62.22%	0.66
backchannel	14412	16277	13291	1121	2986	92.22%	81.66%	0.87
apprec	1926	1816	1312	614	504	68.12%	72.25%	0.70
apology	29	7	5	24	2	17.24%	71.43%	0.28
answer	138	82	31	107	51	22.46%	37.80%	0.28
ans_dispref	109	42	6	103	36	5.50%	14.29%	0.08
agree	4725	3403	2270	2455	1133	48.04%	66.71%	0.56
affirm	376	346	153	223	193	40.69%	44.22%	0.42
acknowledge	636	471	300	336	171	47.17%	63.69%	0.54
abandon	5492	5899	3808	1684	2091	69.34%	64.55%	0.67

Table 4.8: Table showing complete breakdown of the classes in the best run with dependency features.

effect and the margins between the usage of the different dependency parsers are small.

The difference between the two Switchboard systems is also larger in this experiment. It increased from 0.005 percentage points to 0.11 percentage points with the Post-Processed corpus performing the worst.

The baseline still has the lowest performance at 75.15% and there is still a large difference between the parsers using dependency features and the baseline. The difference between the baseline and the Post-Processed system is 0.36 percentage points. This result corroborates with our claim that dependency features can improve the the accuracy of dialog act recognition.

Chapter 5

Conclusion

In this thesis we have presented an algorithm for augmenting the Penn Switchboard Treebanks to incorporate some phenomena found in spoken language and proposed to use them to aid dialog act classification. The theoretical basis for classification and parsing of spoken language was described in Chapter 2. We briefly discussed some of the previous work done in both the field of spoken language parsing and dialog act classification.

In order to represent phenomena characteristic of spoken language in dependency parsers we introduced two new tags "hesitation" and "repair" in Chapter 3. To integrate these labels into a treebank we first converted the Penn Switchboard Treebank to a dependency format using the Stanford Converter. We proposed an algorithm that converts the dependency trees further from the output of the Stanford converter to incorporate the new labels.

The new treebank was used to train a dependency parser which was tested against three other types of parsers, two trained on different version Switchboard and one trained on the Wallstreet Journal. Since the parsers worked on different types of annotation they where not strictly comparable, but we try to mitigate this problem by normalising the datasets based on word-tokens.

Chapter 4 introduces a dialog act classification system with syntactic features extracted from dependency trees. We introduce a baseline based on features that do not use any syntactic information, and a set of features using the dependency representations. The classifier using dependency trees with the new tags was also compared with two other classification systems using dependency parsers trained on a different Switchboard corpora and the Wallstreet Journal. We summarize the findings of these experiments in the following.

There are clear differences between written & spoken language that influences parser results. Testing the different parsers in Chapter 3 on both "WSJ" data and "SWBD" data, we see that there are clear differences. The Switchboard parsers that contain less Wall-Street Journal perform worse on parsing the Wallsteet-Journal. Similarly a parser trained on Wallstreet-

Journal does not perform up to par when parsing Switchboard data.

It is possible to use the Stanford converter to convert Switchboard data and use it as a basis for introducing discourse-related annotation in dependency representations. In Section 3.2.1 we compared two "off-the-shelf" programs for converting the written language parts of the Penn treebanks, the Pennconverter and the Stanford converter. The latter also managed to convert the Switchboard Treebank and keept the repair annotation found in the Switchboard.

Recognizing repairs and hesitations can be done by a dependency parser trained on properly annotated data. Training a Data-Driven Dependency Parser on properly annotated data will enable the parser to recognize repairs and hesitations. Trained on the Swithcboard treebank converted from constituent to dependency and post-processed to introduce such labels, a dependency parser can perform as well as the approach proposed by Charniak and Johnson (2001) on repairs. The dependency parser is able to handle the hesitation phenomena well.

Adding syntactic information in the feature set improves the accuracy of classification in a statistically significant manner when compared to the baseline. All three of our systems with dependency features performed better than the baseline. This means that Using a 15-fold validation scheme and paired T-Test, we showed that the system with depdency information performed significantly better than the baseline.

Some classes need contextual information to be classified more accurately. When expecting the detailed results from our system, we see that a lot of the classes that have low F-Score are classes that would need information outside the utterance to be classified properly. These include classes like summary, third party, repeat and completion.

5.1 Future Work

Since both the conversion of the switchboard to a dependency format and the combination of a dependency parser with a dialog act classification system is fairly new ground, there are quite a number of things that we would have liked to do in order to improve the system. Both areas, the conversion of the Switchboard treebank and the classification system, could have received more attention in the form of refinement and additional experiments.

When it comes to the conversion of the Penn Switchboard Treebank from phrase-structure to dependency trees, the rules that introduce the new labels do not always produce the best trees. This is limited by both the output

of the initial conversion from phrase-structure to dependency trees and the lack of a proper theoretical framework for capturing the repair phenomena in dependency graphs.

The first limitation manifests itself in our use of the Stanford Converter when it places "dep" functions at unexpected locations even when they are not related to the restarts. Looking into improving the Stanford Converter to handle this would perhaps be better than trying to mend the output. This might also be necessary if the linguistic framework incorporates the speech phenomena that we have in the Switchboard Treebank in a manner that requires a better analysis of the original phrase-structure trees.

The repair phenomena cover three phenomena: repairs, duplications and deletions. These phenomena might behave differently in the parser and it would also be interesting to look into using a more fine-grained label set for the different types of phenomena. This would enable us to do a better error analysis of how well the different phenomena is handled by a dependency parser.

On the side of dialog act classification, there are more features that could have been tested. In the baseline model we could have tested the cue-phrase based features proposed by Araki and Webb or tested prosodic features as in the system proposed by Stolcke et al.. In future work, we would also like to test more syntactic features modeling specific phenomena, like question structures. And as we saw that some classes needed more context, it would have been interesting to develop features spanning more than one utterance to capture the classes that need context.

It is not certain that Support Vector Machines and Maltparser were the best choices for this task. We have not done any optimization either on these systems to provide comparisons that are as fair as possible. Both optimizing the current system settings and testing new ones would have been interesting.

It would also have been interesting to explore how the dialog act recognition could be integrated in an end-to-end dialog system with speech recognition errors.

References

Araki, J. (2010). *Dialogue act recognition using cue phrases* (Tech. Rep.). Stanford University, Computer Science Department.

Austin, J. L. (1975). *How to do things with words* (Vol. 1955). Oxford university press.

Bobrow, D. G., Kaplan, R. M., Kay, M., Norman, D. A., Thompson, H., Winograd, T. (1977). Gus, a frame-driven dialog system. *Artificial Intelligence*, *8*(2), 155 - 173.

Bunt, H. (1995). Dynamic interpretation and dialogue theory. *The structure of multimodal dialogue*, *2*, 1–8.

Calhoun, S., Carletta, J., Brenier, J. M., Mayo, N., Jurafsky, D., Steedman, M., Beaver, D. (2010). The nxt-format switchboard corpus: a rich resource for investigating the syntax, semantics, pragmatics and prosody of dialogue. *Language Resources and Evaluation*, *44*(4), 387–419.

Cassel, S. (2009). *Maltparser and liblinear – transition-based dependency parsing with linear classification for feature model optimization.* Master's thesis, Uppsala University.

Chang, C.-C., Lin, C.-J. (2011). LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, *2*, 27:1–27:27. (Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`)

Charniak, E., Johnson, M. (2001). Edit detection and parsing for transcribed speech. In *Proceedings of the second meeting of the north american chapter of the association for computational linguistics on language technologies* (pp. 1–9). Stroudsburg, PA, USA: Association for Computational Linguistics.

Core, M., Allen, J. (1997). Coding dialogs with the damsl annotation scheme. In *Aaai fall symposium on communicative action in humans and machines* (pp. 28–35).

Cortes, C., Vapnik, V. (1995, September). Support-vector networks. *Machine Learning*, *20*(3), 273–297.

Debusmann, R. (2000). An introduction to dependency grammar. *Hausarbeit fur das Hauptseminar Dependenzgrammatik SoSe*, *99*, 1–16.

De Marneffe, M.-C., MacCartney, B., Manning, C. D., et al. (2006). Generating typed dependency parses from phrase structure parses. In *Proceedings*

of the international conference on language resources (lrec) (Vol. 6, pp. 449–454).

De Marneffe, M.-C., Manning, C. D. (2008). *Stanford typed dependencies manual* (Tech. Rep.). Stanford University.

Fernández, R. (2006). *Non-sentential utterances in dialogue: Classification, resolution and use.* Phd thesis, King's College London.

Gaifman, H. (1965). Dependency systems and phrase-structure systems. *Information and control*, *8*(3), 304–337.

Hall, J. (2008). *Transition-based natural language parsing with dependency and constituency representations.* Phd thesis, Växjö University.

Hays, D. G. (1964). Dependency theory: A formalism and some observations. *Language*, *40*(4), 511–525.

Johansson, R., Nugues, P. (2007, May 25-26). Extended constituent-to-dependency conversion for English. In *Proceedings of nordic conference of computational linguistics (nodalida) 2007* (p. 105-112).

Jurafsky, D., Martin, J. H. (2009). *Speech and language processing (2nd edition).* Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Jurafsky, D., Shriberg, E., Biasca, D. (1997). Switchboard swbd-damsl shallow-discourse-function annotation coders manual. *Institute of Cognitive Science Technical Report*, 97–02.

Jørgensen, F. (2007). The effects of disfluency detection in parsing spoken language. In *Proceedings of the 16th nordic conference of computational linguistics (nodalida)* (pp. 240–244).

Marsland, S. (2009). *Machine learning: An algorithmic perspective* (1st ed.). Chapman & Hall/CRC.

McDonald, R., Pereira, F., Ribarov, K., Hajič, J. (2005). Non-projective dependency parsing using spanning tree algorithms. In *Proceedings of the conference on human language technology and empirical methods in natural language processing* (pp. 523–530).

Meteer, M. W., Taylor, A. A., MacIntyre, R., Iyer, R. (1995). *Dysfluency annotation stylebook for the switchboard corpus.* University of Pennsylvania.

Mitchell, T. M. (1997). *Machine learning.* McCraw-Hill Science/Engineering/Math.

Nivre, J. (2005). *Dependency grammar and dependency parsing* (Tech. Rep.). Växjö University.

Nivre, J. (2006). Two strategies for text parsing. *SKY Journal of Linguistics*, *19*, 440–448.

Nivre, J. (2007). Dependency parsing of spoken swedish. In *Communication – action – meaning. a festschrift to jens allwood.* (pp. 203–211). Göteborg University: Department of Linguistics.

Nivre, J., Hall, J. (2010). *A quick guide to maltparser optimization* (Tech. Rep.).

Nivre, J., Hall, J., Nilsson, J., Chanev, A., Eryigit, G., Kübler, S., ... Marsi, E. (2007). Maltparser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering*, *13*(2), 95–135.

Pereira, F. C., Warren, D. H. (1980). Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Artificial intelligence*, *13*(3), 231–278.

Pieraccini, R., Levin, E. (1992). Stochastic representation of semantic structure for speech understanding. *Speech Communication*, *11*(2–3), 283 - 288.

Rehbein, I., Schalowski, S., Wiese, H. (2012). Annotating spoken language. In *Best practices for speech corpora in linguistic research workshop programme* (p. 29).

Sacks, H., Schegloff, E. A., Jefferson, G. (1974). A simplest systematics for the organization of turn-taking for conversation. *Language*, 696–735.

Schalkwyk, J., Beeferman, D., Beaufays, F., Byrne, B., Chelba, C., Cohen, M., ... Strope, B. (2010). "your word is my command": Google search by voice: A case study. In A. Neustein (Ed.), *Advances in speech recognition* (p. 61-90). Springer US.

Searle, J. R. (1976, 4). A classification of illocutionary acts. *Language in Society*, *5*, 1–23.

Stent, A., Dowding, J., Gawron, J. M., Bratt, E. O., Moore, R. (1999). The commandtalk spoken dialogue system. In *Proceedings of the 37th annual meeting of the association for computational linguistics on computational linguistics* (pp. 183–190).

Stolcke, A., Ries, K., Coccaro, N., Shriberg, E., Bates, R., Jurafsky, D., ... Meteer, M. (2000). Dialogue act modeling for automatic tagging and recognition of conversational speech. *Computational linguistics*, *26*(3), 339–373.

Traum, D. R. (1991). *Towards a computational theory of grounding in natural language conversation* (Tech. Rep.). DTIC Document.

Traum, D. R., Allen, J. F. (1992). A "speech acts" approach to grounding in conversation. In *In proceedings of international conference on spoken language processing (icslp'92* (pp. 137–140).

Van Noord, G., Bouma, G., Koeling, R., Nederhof, M.-J. (1999). Robust grammatical analysis for spoken dialogue systems. *Natural language engineering*, *5*(01), 45–93.

Webb, N. (2010). *Cue-based dialogue act classification*. Phd thesis, University of Sheffield.

Young, S. (2002). *The statistical approach to the design of spoken dialogue systems* (Tech. Rep.). Cambridge University Engineering Department.